REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-98-

0366

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE
	July 1996	Final

4. TITLE AND SUBTITLE
Retiming, Folding, and Register Minimization for DSP Synthesis

5. FUNDING NUMBERS

6. AUTHORS
Tracy Carroll Denk

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Minnesota

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NI
110 Duncan Avenue, Room B-115
Bolling Air Force Base, DC 20332-8080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
See attached.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Abstract

This thesis introduces some formal techniques which can be used for synthesis of VLSI (very large scale integration) architectures for DSP (digital signal processing) algorithms. These techniques can be used to design architectures for single-rate/single-dimensional DSP, multirate/single-dimensional DSP, and single-rate/multi-dimensional DSP.

For single-rate/single-dimensional DSP, we have developed a novel technique for exhaustively generating all retiming and scheduling solutions for the DSP algorithm. The significance of this contribution is two-fold. First, it allows a circuit designer to explore a large space of possible high-level implementations for the algorithm, which allows the designer to make a good decision about the high-level architectural details of the design. Second, this work explicitly shows the important interaction between retiming and scheduling in high-level synthesis. While retiming and scheduling have been treated as separate problems in the past, our work uses a mathematical framework to show that retiming is a special case of scheduling.

Also for single-rate/single-dimensional DSP, we have developed techniques for computing the minimum number of registers required to implement a statically scheduled DSP program. Closed-form expressions are derived for computing the minimum number of registers assuming various memory models with or without retiming the scheduled DFG. This is an important problem because memory typically occupies a large portion of the area of a DSP implementation (often over half of the area), and minimizing this area leads to more efficient designs.

For multirate/single-dimensional DSP, we have developed a multirate folding technique which can be used to synthesize single-rate architectures from multirate DSP algorithms. Prior to the development of this formal technique, the design of single-rate architectures for multi-rate DSP algorithms was performed using *ad hoc* design techniques.

For single-rate/multi-dimensional DSP, we have developed two techniques for retiming two-dimensional data-flow graphs. These techniques are designed to minimize the memory requirements under a given clock period constraint. These techniques can result in retimed circuits which use less than 50% of the memory required by previously used techniques.

UNIVERSITY OF MINNESOTA

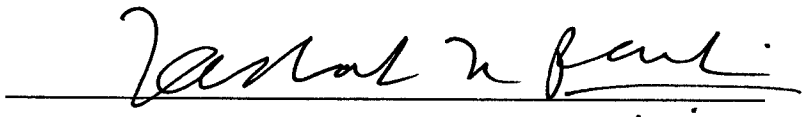
This is to certify that I have examined this bound copy of a doctoral thesis by

Tracy Carroll Denk

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Keshab K. Parhi

Name of Faculty Adviser



Signature of Faculty Adviser

July 5, 1996

Date

GRADUATE SCHOOL

Retiming, Folding, and Register Minimization for DSP Synthesis

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Tracy Carroll Denk

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

July 1996

Acknowledgement

I wish to thank my advisor, Professor Keshab K. Parhi, for his guidance, patience, and support since my days as an undergraduate at Minnesota. I am grateful for the many valuable lessons I have learned from him throughout the years. It has certainly been a privilege and a pleasure to work with Dr. Parhi. I would also like to thank the members of my examining committee – Professor Cherkassky, Professor Tewfik, Professor Carlis, and Professor Papanikolopoulos – for supporting my work.

I would like to thank the Air Force and the Advanced Research Projects Agency for supporting me in parts through an Air Force Laboratory Graduate Fellowship and a grant from the Advanced Research Projects Agency and the Solid State Electronics Directorate, Wright-Patterson AFB (contract number AF/F33615-93-C-1309).

I would like to thank several people for their help with the work in this thesis. I thank Dr. Ching-Yi Wang and Dr. Kazuhito Ito for generating schedules for the register minimization work. I thank John Bratt for many helpful discussions on the exhaustive retiming and scheduling work, and I thank Mayukh Majumdar for helping to work out some of the two-dimensional retiming details. I also thank Dr. Ching-Yi Wang and Chong Xu for helpful discussions on the topic of multirate folding. The help of these and several other talented people made working on this thesis a very enjoyable experience.

I am very grateful to my parents, to whom this thesis is dedicated, for their abundant love, guidance, encouragement, and support throughout the years. I will never be able to thank them enough. Hats off to you, Bruce and Cindy! I am also grateful to the rest of my family for their support. I am particularly glad that my grandparents, RC and Shirley, are able to share the joy of this accomplishment with me.

I thank my friends for the many fond memories and for making sure that I didn't spend all of my time studying.

I thank God for everything.

Abstract

This thesis introduces some formal techniques which can be used for synthesis of VLSI (very large scale integration) architectures for DSP (digital signal processing) algorithms. These techniques can be used to design architectures for single-rate/single-dimensional DSP, multirate/single-dimensional DSP, and single-rate/multi-dimensional DSP.

For single-rate/single-dimensional DSP, we have developed a novel technique for exhaustively generating all retiming and scheduling solutions for the DSP algorithm. The significance of this contribution is two-fold. First, it allows a circuit designer to explore a large space of possible high-level implementations for the algorithm, which allows the designer to make a good decision about the high-level architectural details of the design. Second, this work explicitly shows the important interaction between retiming and scheduling in high-level synthesis. While retiming and scheduling have been treated as separate problems in the past, our work uses a mathematical framework to show that retiming is a special case of scheduling.

Also for single-rate/single-dimensional DSP, we have developed techniques for computing the minimum number of registers required to implement a statically scheduled DSP program. Closed-form expressions are derived for computing the minimum number of registers assuming various memory models with or without retiming the scheduled DFG. This is an important problem because memory typically occupies a large portion of the area of a DSP implementation (often over half of the area), and minimizing this area leads to more efficient designs.

For multirate/single-dimensional DSP, we have developed a multirate folding technique which can be used to synthesize single-rate architectures from multirate DSP algorithms. Prior to the development of this formal technique, the design of single-rate architectures for multi-rate DSP algorithms was performed using *ad hoc* design techniques.

For single-rate/multi-dimensional DSP, we have developed two techniques for retiming two-dimensional data-flow graphs. These techniques are designed to minimize the memory requirements under a given clock period constraint. These techniques can result in retimed circuits which use less than 50% of the memory required by previously used techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This thesis introduces some formal techniques which can be used for the synthesis of VLSI [1, 2] (very large scale integration) architectures for DSP [3, 4, 5, 6] (digital signal processing) algorithms. DSP is used in many applications such as compact disc players, digital television, videoconferencing systems, digital telephony, radar, and sonar, just to name a few. VLSI architectures for DSP algorithms must be designed to satisfy constraints on the sampling rate, chip size, and power consumption. Without adequate implementations, DSP algorithms would not be useful to consumers.

Figure 1.1 shows a simplified version of the process of generating a silicon solution for a given application. There are three main steps in this process. The first step is to develop or choose the proper DSP algorithm for the application. The second step is high-level synthesis [7] -[26], which maps the algorithm to a VLSI architecture, and the third step is low-level synthesis, which maps the VLSI architecture to silicon. These three steps are not independent, and it has become apparent that a good understanding of all three of these steps is required to design an efficient silicon solution for a given application. The focus of this thesis, as indicated in the figure, is on the area of high-

level synthesis, i.e., designing high-level VLSI architectures for DSP algorithms. The formal techniques introduced in this thesis help provide a better understanding of the algorithm $\rightarrow$ architecture step and provide new techniques for mapping algorithms to architectures.

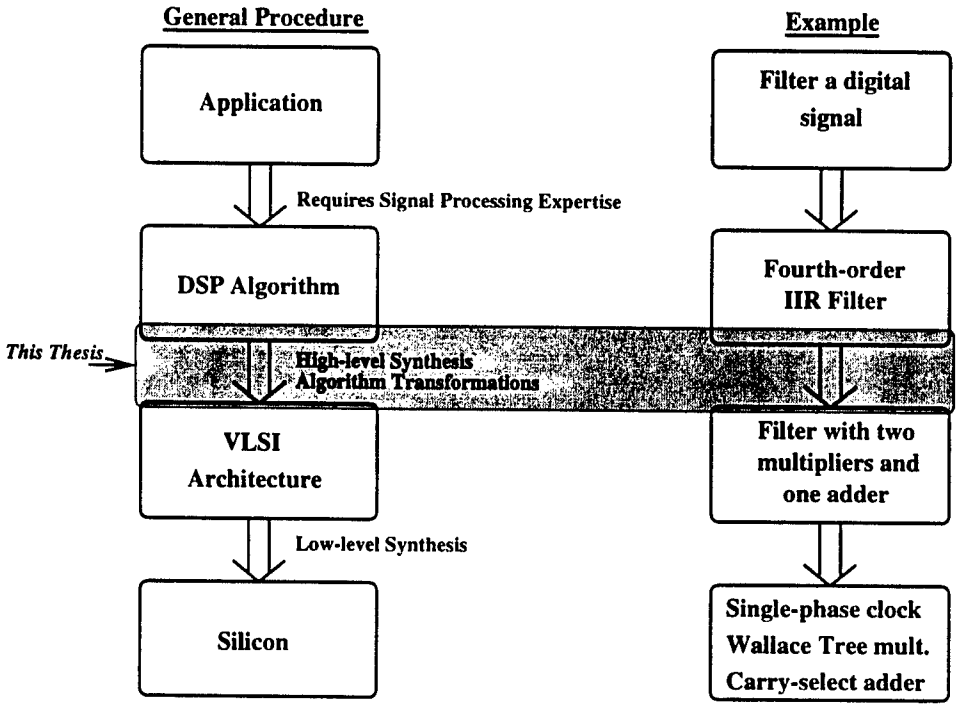


Figure 1.1: A simplified version of the design process from application to silicon.

As DSP algorithms become more complex and transistor sizes become smaller, the tasks of designing and testing VLSI architectures for DSP have become very challenging due to the sheer size of these tasks. In order for products to be introduced in a timely manner, CAD (computer-aided design) tools [8, 26, 24, 16, 10, 12, 20, 22, 23, 14, 15] are often required. These tools not only decrease design time, but they also make the design process more tractable, improving the reliability of the final VLSI design. These CAD tools are based on formal design techniques which can be used to automate the process of synthesizing VLSI architectures for DSP algorithms.

Some formal techniques for synthesizing VLSI architectures for DSP algorithms are introduced in this thesis. These techniques can be used to explore new VLSI designs for DSP algorithms and improve CAD tools which are used to design VLSI architectures for DSP algorithms. A description of these techniques is given in the following section.

1.2 Contributions

The contributions of this thesis fall into the categories of retiming [27], folding [28], and register minimization [29]. A concise description of these contributions follows.

- Retiming
  - Exhaustive retiming: A novel technique for exhaustively generating all retiming solutions for a DFG is developed. This technique, which is based on the ideas in [30], [31], allows a circuit designer to examine many retiming solutions rather than a single solution which is generated using a heuristic or an optimization scheme. This is useful because it is easy to select the best retimed solution optimized for circuit parameters, such as routing area, from all retiming solutions.
  - Two-dimensional retiming: Two novel techniques are developed for retiming two-dimensional data-flow graphs (DFGs) to minimize the memory requirements under a given clock period constraint. These two techniques are *integer linear programming (ILP) 2-D retiming* and *orthogonal 2-D retiming* [32]. These techniques offer greater flexibility than the technique proposed in [33], and they can reduce the memory requirement of retimed circuits by over 50% compared to the technique in [34].

- Multirate retiming: Multirate retiming constraints are formalized as part of the multirate folding formulation. Multirate retiming has received little attention in the past, and most of the previous work has been focused on maintaining properties such as liveness and reachability in synchronous data-flow graphs (e.g., see [35]). The treatment of multirate retiming in this thesis considers the problem at a more fundamental level by using some simple identities of multirate DSP [5]. We show that our multirate retiming formulation is useful for high-level synthesis of single-rate VLSI architectures for multirate DSP algorithms [36].

- Folding

  - Exhaustive Scheduling: A novel technique for exhaustively generating all time schedules for folding a DFG is developed [31]. This technique, termed "exhaustive scheduling", has three important features. First, it shows the important interaction between retiming and scheduling in a solid mathematical framework. Retiming and scheduling have only recently been considered together [11, 26, 12, 37, 38], and none of these works has given a mathematical framework for demonstrating how retiming and scheduling interact in high-level synthesis. Second, our mathematical framework can be used to show that retiming is simply a special case of scheduling. Many researchers have thought this to be true for a long time, but none have shown this mathematically. Finally, exhaustive scheduling allows a circuit designer the option of evaluating several different schedules for characteristics that are difficult to include in heuristics [12, 15, 26] or ILP models [39, 40, 22, 37] used for scheduling.

- Multirate folding: A novel technique for folding multirate DSP algorithms is developed [36]. This technique maps multirate DSP algorithms to single-rate VLSI architectures. For example, multirate folding can be used to design single-rate architectures for algorithms which use multirate filter banks, such as the discrete wavelet transform (DWT) [41, 42, 43, 44, 45]. Prior to the development of multirate folding, single-rate VLSI architectures for multirate DSP algorithms were designed using *ad hoc* design techniques. Multirate folding provides a vehicle for systematically designing improved architectures for multirate DSP algorithms.

- Register Minimization

  - Single-rate register minimization: Expressions are derived for computing the minimum number of registers required to implement a statically scheduled single-rate DSP algorithm [46]. To the best of our knowledge, no such expressions existed prior to this work. Expressions are derived for three different memory models. These expressions can be used in CAD tools to evaluate the quality of schedules with respect to memory requirements. For example, these expressions are used along with our exhaustive scheduling technique to determine the schedules which require the minimum number of registers.

  - Multirate register minimization: Expressions are derived for computing the minimum number of registers required to implement a statically scheduled multirate DSP algorithm. This novel approach to evaluating memory requirements allows for the design of memory-efficient single-rate architectures for the implementation of multirate DSP algorithms.

1.3 Outline

This thesis is organized as follows. The exhaustive retiming and scheduling algorithms are developed in Chapter 2. This chapter also provides a background information on retiming and folding. Register minimization for statically scheduled single-rate data-flow graphs is considered in Chapter 3. Chapter 4 contains the derivation of the multirate folding transformation, including the work on retiming for multirate folding and register minimization for folded multirate DSP algorithms. The two-dimensional retiming techniques are derived in Chapter 5, and conclusions and suggestions for future research are presented in Chapter 6.

Chapter 2

Exhaustive Retiming and Scheduling

2.1 Introduction

Time scheduling and retiming [27] are important tools used to map behavioral descriptions of algorithms to physical realizations. These tools are used during the design of software for programmable digital signal processors (DSPs), during high-level synthesis of applications-specific integrated circuits (ASICs), and during the design of reconfigurable hardware such as field-programmable gate arrays (FPGAs). Time scheduling and retiming operate directly on a behavioral description of the algorithm, such as a data-flow graph (DFG). Since the decisions made at the algorithmic level tend to have greater impact on the design than those made at lower levels, the importance of time scheduling and retiming cannot be overstated.

This chapter presents new formulations of the time scheduling and retiming problems, and based on these formulations, new techniques are developed to determine the solutions to these problems [31]. (From this point forward, we shall refer to *time scheduling* as simply *scheduling.*) These formulations are valid for strongly connected (SC) graphs, where a strongly connected graph has a path $u \rightsquigarrow v$ and a path $v \rightsquigarrow u$ for every pair of

nodes u, v in the graph. We focus on strongly connected graphs because these graphs traditionally present the greatest challenges when they are mapped to physical realizations due to the feedback present in the graphs. An example of a strongly connected DFG is the fifth-order wave digital elliptic filter [47] in Figure 2.18 which is commonly used as a benchmark for demonstrating high-level synthesis techniques.

Scheduling consists of assigning execution times to the operations in a DFG such that the precedence constraints of the DFG are not violated. A great deal of literature exists on the topic of scheduling in the context of high-level synthesis for ASIC design for DSP applications [7] -[26]; however, none of these works gives a formal definition of scheduling along with systematic techniques for exhaustively generating the solutions to the scheduling problem. This chapter presents new scheduling formulations and algorithms for exhaustively generating the solutions to the scheduling problem. Two scheduling problems are considered, namely, scheduling for time-multiplexed execution on bit parallel architectures and scheduling for execution on bit-serial architectures.

Retiming consists of moving delays around in a DFG without changing its functionality. As with scheduling, there is a huge body of literature on retiming, and new applications for retiming are constantly being found. For example, due to the recent demand for low-power digital circuits in portable devices, some recent work has focused on retiming for power minimization [48]. The groundbreaking paper on retiming [27] describes algorithms for tasks such as retiming to minimize the clock period and retiming to minimize the number of registers (states) in the retimed circuit. An approach to retiming which is based on circuit theory can be used to generate all retiming solutions for a DFG [30]. This approach was the motivation for our work on exhaustive scheduling. In this chapter, we show that retiming is a special case of scheduling, and consequently, the formulation of the scheduling problem and the techniques for exhaustively generating

the scheduling solutions can also be applied to retiming.

The impact of the formulations derived in this chapter are as follows.

- The interaction between retiming and scheduling is important [11], and our formulations give a simple way to observe this interaction.
- We show that retiming is a special case of scheduling.
- We give solid mathematical descriptions of the scheduling and retiming problems in a common framework.
- We develop techniques for generating all solutions to a particular scheduling or retiming problem. This allows a developer the ability to search the design space for the best solution, particularly when various parameters are difficult to model and include in a cost function. This has applications to software design, ASIC design, and design for reconfigurable hardware implementations.
- Our formulations provide for a better understanding of scheduling and retiming which can be used to develop new heuristics for these problems.

Many of the results in this chapter rely upon graph theory. Section 2.2 gives a review of some results from graph theory along with the derivation of an algorithm for finding the independent loops in a strongly connected directed graph. Our formulations for scheduling to bit-parallel and bit-serial architectures are given in Section 2.3 along with an explanation of how retiming can be viewed as a special case of scheduling. Section 2.4 contains the description of a systematic technique used to exhaustively generate the scheduling and retiming solutions. Section 2.5 describes two techniques for exhaustively generating the schedules which satisfy a given set of resource constraints for a bit-parallel

architecture. Section 2.5 includes the results of scheduling the fifth-order wave-digital elliptic filter in Figure 2.18 with and without resource constraints. Our conclusions are given in Section 2.6.

2.2 Introduction to Graph Theory

This section provides a brief introduction to graph theory followed by an algorithm for finding the independent loops in a strongly connected directed graph. Most of the definitions and results in Sections 2.2.1 and 2.2.2 can be found in [49].

2.2.1 Basic Definitions

We are concerned only with *directed graphs*. A directed graph G is represented as $G =< V, E, d, w >$, where

- V is the set of vertices (nodes) of G. The vertices represent computations.
- E is the set of directed edges of G. A directed edge $e \in E$ from node $u \in V$ to node $v \in V$ is denoted as $u \xrightarrow{e} v$. The edges represent communication between the nodes.
- $w(e)$ is the number of delays on the edge e, also referred to as the *weight* of the edge.
- $d(v)$ is the computation time of the node v.

A directed path $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots \xrightarrow{e_{n-1}} v_{n-1} \xrightarrow{e_n} v_n$ is denoted as $v_0 \rightsquigarrow v_n$. A simple path is a path with distinct edges, and an elementary path has distinct nodes. A cycle is a closed path (i.e., $v_o = v_n$). A simple cycle has distinct edges and an elementary cycle has

distinct nodes. An elementary cycle in a directed graph will be referred to as a "loop" in this chapter.

A directed graph is *strongly connected* if for every pair of vertices $u, v \in V$, there exists a path $u \rightsquigarrow v$ and $v \rightsquigarrow u$. A directed spanning tree is a subgraph of G which has a root node v_R and a path $v_R \rightsquigarrow v$ for all $v \in V$ except v_R. The directed spanning tree contains no cycles. If $|V|$ is the number of nodes in G, then a directed spanning tree contains exactly $|V|$ nodes and $|V| - 1$ edges. An edge of a directed spanning tree is called a branch, and the edges of G not included in the tree are called links. Every strongly connected graph contains a directed spanning tree.

An edge e from u to v ($u \xrightarrow{e} v$) is incident with vertices u and v. More specifically, e is incident *from* u and incident *into* v.

The set operations such as union, intersection, difference, complement, etc., are operations on the *edges* of a graph. Let G_a and G_b be two subgraphs of a connected graph G. $G_a \cup G_b$ consists of all edges in G_a or G_b (or both) and the vertices incident with these edges. $G - G_a$ is formed by removing all edges in G_a from G, and then removing all vertices with no incident edges.

2.2.2 Matrix Representations

A strongly connected graph contains exactly $|E| - |V| + 1$ linearly independent loops (this is shown in Section 2.2.3). Let $\mathbf{B}$ be the *fundamental loop matrix*. This matrix, which has dimensions $(|E| - |V| + 1) \times |E|$, is defined as

$$b_{ij} = \begin{cases} 1 & \text{if edge } j \text{ is in loop } i \\ 0 & \text{otherwise} \end{cases} .$$

Each row of $\mathbf{B}$ represents one of $|E| - |V| + 1$ linearly independent loops in $\mathbf{B}$.

Let $\mathbf{A}$ be the *oriented incidence matrix* of G. This matrix, which has dimensions

$|V| \times |E|$, is defined as

$$a_{ij} = \begin{cases} 1 & e_j \text{ is incident from } v_i \\ -1 & e_j \text{ is incident into } v_i \\ 0 & e_j \text{ and } v_i \text{ are not incident} \end{cases}$$

and $\text{rank}(\mathbf{A}) = |V| - 1$. The *reduced oriented incidence matrix* $\mathbf{A}_R$ is defined to be any $|V| - 1$ rows of $\mathbf{A}$. $\mathbf{A}_R$ has dimensions $(|V| - 1) \times |E|$ and $\text{rank}(\mathbf{A}_R) = |V| - 1$.

Two important relationships between the fundamental loop matrix and the oriented incidence matrix are $\mathbf{B}\mathbf{A}^T = \mathbf{0}$ and $\mathbf{B}\mathbf{A}_R^T = \mathbf{0}$.

Example 2.1 *Consider the directed graph in Figure 2.1. This graph has six nodes and nine edges ($|V| = 6$ and $|E| = 9$). The branches of a directed spanning tree are shown with solid lines and the links are shown with dashed lines. The spanning tree contains $|V| - 1$ edges and $|V|$ nodes. One possibility for the $((|E| - |V| + 1) \times |E|) = (4 \times 9)$ $\mathbf{B}$ matrix is*

$$\mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}, \tag{2.1}$$

whose columns and rows appear according to the numbering of the edges and loops, respectively, in Figure 2.1. $\mathbf{A}$ *is the $(|V| \times |E|) = (6 \times 9)$ matrix*

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 1 & 0 & -1 & 0 \\ -1 & 1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The reader can verify that $\text{rank}(\mathbf{A}) = |V| - 1 = 5$ *and* $\mathbf{B}\mathbf{A}^T = \mathbf{0}_{4\times 6}$. *One possible reduced incidence matrix is the $((|V| - 1) \times |E|) = (5 \times 9)$ matrix*

$$\mathbf{A}_R = \begin{bmatrix} 0 & -1 & 1 & 0 & 0 & 1 & 0 & -1 & 0 \\ -1 & 1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.2}$$

which is simply $\mathbf{A}$ *with the the first row (the row corresponding to node* 1*) removed. The reader can verify that* $\text{rank}(\mathbf{A}_R) = |V| - 1 = 5$ *and* $\mathbf{B}\mathbf{A}_R^T = \mathbf{0}_{4\times 5}$.

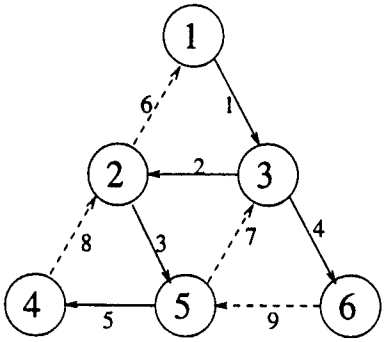


Figure 2.1: A strongly connected graph. The branches of a spanning tree are shown with solid lines, while the links of the corresponding cotree are shown with dashed lines.

2.2.3 Finding the Independent Loops of a Strongly Connected Graph

Recall that the fundamental loop matrix $\mathbf{B}$ has $|E| - |V| + 1$ rows, each of which corresponds to an independent loop. This section gives an algorithm for finding $|E| - |V| + 1$ independent loops of a strongly connected graph. Let G_T be a directed spanning tree of G, where v_R is the root node of G_T, i.e., there is a path $v_R \rightsquigarrow v$ for all $v \in V$ except v_R. Algorithm FFL (Find Fundamental Loops) is given below.

Algorithm FFL (Find Fundamental Loops)

$G_R^{(1)} = v_R$;

FOR $(k = 1$ TO $|E| - |V| + 1)$

{

STEP 1: l_k = a link in $(G - G_R^{(k)})$ which is incident to $G_R^{(k)}$;

STEP 2: loop(k) = A loop in $G_T \cup G_R^{(k)} \cup l_k$ which contains l_k;

STEP 3: $G_R^{(k+1)} = G_R^{(k)} \cup$ loop(k);

}

The $|E|-|V|+1$ loops denoted as loop(k), $1 \le k \le (|E|-|V|+1)$, are the fundamental

loops of G.

Algorithm FFL maintains a subgraph G_R which initially consists of the root node of the directed spanning tree G_T. During iteration k, a link l_k in $(G - G_R^{(k)})$ which is incident into a node in $G_R^{(k)}$ is chosen in STEP 1. This link, along with edges in $G_T \cup G_R^{(k)}$, form a loop which we denote as loop(k). $G_R^{(k)}$ is then updated at the end of the iteration.

To prove that Algorithm FFL works, we need to show that link l_k in STEP 1 exists for each iteration $1 \leq k \leq (|E| - |V| + 1)$, and we need to show that loop(k) in STEP 2 exists for $1 \leq k \leq (|E| - |V| + 1)$.

The following three lemmas are used to prove that link l_k exists in STEP 1 of Algorithm FFL.

Lemma 2.1 *$G_R^{(k)}$ is strongly connected (SC).*

Proof: By induction. $G_R^{(1)} = v_R$ is SC. Assume that $G_R^{(k)}$ is SC. Each vertex in $(G_R^{(k+1)} - G_R^{(k)})$ is part of loop(k) which has at least one vertex in $G_R^{(k)}$, so $G_R^{(k+1)}$ is also SC. □

Lemma 2.2 *For every node v in $G_R^{(k)}$ except v_R, there is a branch of G_T in $G_R^{(k)}$ which is incident into the node v.*

Proof: By induction. This holds for $G_R^{(1)}$. Assume this holds for $G_R^{(k)}$. All edges of loop(k) are in $G_T \cup G_R^{(k)} \cup l$. Since $G_R^{(k+1)} = G_R^{(k)} \cup$ loop(k), all edges in $(G_R^{(k+1)} - G_R^{(k)})$ except l_k are tree branches. Since l_k is incident into a node in $G_R^{(k)}$, each node in $G_R^{(k+1)}$ but not in $G_R^{(k)}$ must have a tree branch in $G_R^{(k+1)}$ incident into it. So every node in $G_R^{(k+1)}$, except v_R, has a tree branch in $G_R^{(k+1)}$ incident into it. □

The following lemma uses the result of Lemma 2.2.

Lemma 2.3 *There are no branches of G_T in $(G - G_R^{(k)})$ which are incident to a node in $G_R^{(k)}$.*

Proof: By contradiction. Assume a branch exists in $(G - G_R^{(k)})$ which is incident into the node v in $G_R^{(k)}$. Then v must have two incident branches because we know from Lemma 2.2 that there is also a branch in $G_R^{(k)}$ which is incident into v. However, no node can have two incident branches because multiple paths $v_R \rightsquigarrow v$ would exist in G_T, which is not allowed. □

Lemma 2.1 and Lemma 2.3 are used to prove that l_k exists in STEP 1 of Algorithm FFL.

Theorem 2.4 *Link l_k in STEP 1 of Algorithm FFL exists for all iterations $1 \leq k \leq (|E| - |V| + 1)$.*

Proof: $(G - G_R^{(k)})$ contains exactly $|E| - |V| + 2 - k$ links at the start of iteration k, so $(G - G_R^{(k)})$ contains at least one link during each iteration. Consider the following two cases:

1. There exists a node $v \in V$ which is not in $G_R^{(k)}$, i.e., no edges in $G_R^{(k)}$ are incident into or from v. Since G is SC, there is a path from v to v_R, implying that there is a path from v to $G_R^{(k)}$. According to Lemma 2.3, there are no branches in $(G - G_R^{(k)})$ which are incident to a node in $G_R^{(k)}$, so there must be a link in $(G - G_R^{(k)})$ which is incident into $G_R^{(k)}$ allowing a path to exist from v to $G_R^{(k)}$.

2. $G_R^{(k)}$ contains all nodes. Each link in $(G - G_R^{(k)})$ is incident into $G_R^{(k)}$ in this case.

□

The following theorem uses Lemma 2.1 to show that loop(k) in STEP 2 exists for $1 \leq k \leq (|E| - |V| + 1)$.

Theorem 2.5 *There is a loop containing l_k in $G_T \cup G_R^{(k)} \cup l_k$.*

Proof: Consider Figure 2.2. Nodes v_R and v_{IN} are in $G_R^{(k)}$. Link l_k is in $(G - G_R^{(k)})$. Path p_2 exists in $G_R^{(k)}$ because $G_R^{(k)}$ is SC (according to Lemma 2.1). Path p_1 exists in G_T because v_R is the root of the directed spanning tree. So a directed cycle $v_X \xrightarrow{l_k} v_{IN} \rightsquigarrow v_R \rightsquigarrow v_X$ exists in $G_T \cup G_R^{(k)} \cup l_k$. If this directed cycle is not elementary, then it must have the form $v_X \xrightarrow{l_k} v_{IN} \rightsquigarrow v_{COMMON} \rightsquigarrow v_R \rightsquigarrow v_{COMMON} \rightsquigarrow v_X$, from which the elementary directed cycle (loop) $v_X \xrightarrow{l_k} v_{IN} \rightsquigarrow v_{COMMON} \rightsquigarrow v_X$ can be found. □

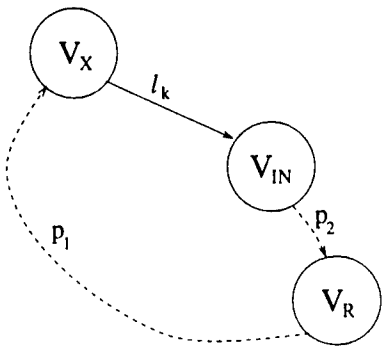


Figure 2.2: A directed cycle created by adding link l_k which goes from $(G - G_R^{(k)})$ to $G_R^{(k)}$.

We construct the fundamental loop matrix **B** by letting loop(k) from Algorithm FFL be the k-th row of **B**. The edges in the graph are numbered such that the first $(|V| - 1)$ columns of **B** correspond to the branches of the spanning tree of G, and the remaining $(|E| - |V| + 1)$ columns correspond to the links. The link l_k is assigned to the $(|V| - 1 + k)$-th column of **B**. By constructing the fundamental loop matrix in this manner, it has the form

$$\mathbf{B} = \left[\begin{array}{c|c} \mathbf{C} & \mathbf{L} \end{array} \right], \tag{2.3}$$

where $\mathbf{C}$ is an $(|E|-|V|+1)\times(|V|-1)$ matrix and $\mathbf{L}$ is an $(|E|-|V|+1)\times(|E|-|V|+1)$ lower triangular matrix with ones on the diagonal. Note that the columns of $\mathbf{L}$ correspond to the links of G while the columns of $\mathbf{C}$ correspond to the branches of G. Because of its form, $\mathbf{B}$ has rank $(|E|-|V|+1)$.

It can also be shown that adding more loops of G to $\mathbf{B}$ (adding a loop would consist of adding a row to $\mathbf{B}$) does not increase its rank. Therefore, the $(|E|-|V|+1)$ rows of $\mathbf{B}$ form a basis for the loops of G.

Example 2.2 *This example uses Algorithm FFL to form the fundamental loop matrix for the graph in Figure 2.1. The spanning tree with node* 1 *as the root node is shown in Figure 2.3(a). At the start of Algorithm FFL* $G_R^{(1)}$ *is node 1. During iteration* $k=1$*, the only possibility for link* l_1 *is edge 6. The only possibility for loop*(1) *is* $1 \xrightarrow{1} 3 \xrightarrow{2} 2 \xrightarrow{6} 1$. $G_R^{(2)}$ *is circled in Figure 2.3(b). During iteration* $k=2$*, there are two possibilities for link* l_2*, namely, edges 7 and 8. Choosing edge 7 as* l_2 *results in loop*$(2) = 3 \xrightarrow{2} 2 \xrightarrow{3} 5 \xrightarrow{7} 3$. $G_R^{(3)}$ *is circled in Figure 2.3(c). During iteration* $k=3$*, the two possibilities for link* l_3 *are edges 8 and 9. Choosing edge 8 as* l_3 *results in loop*$(3) = 2 \xrightarrow{3} 5 \xrightarrow{5} 4 \xrightarrow{8} 2$. $G_R^{(4)}$ *is circled in Figure 2.3(d). During iteration* $k=4$*, link* l_4 *is edge 9, and loop*(4) *is* $3 \xrightarrow{4} 6 \xrightarrow{9} 5 \xrightarrow{7} 3$. *The fundamental loop matrix is*

$$\mathbf{B} = \left[\begin{array}{ccccc|cccc} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}\right],$$

Note that $\mathbf{B}$ *has the desired form as given in (2.3). Row* k *corresponds to loop*(k) *from Algorithm FFL and column* i *corresponds to edge* i *of* G.

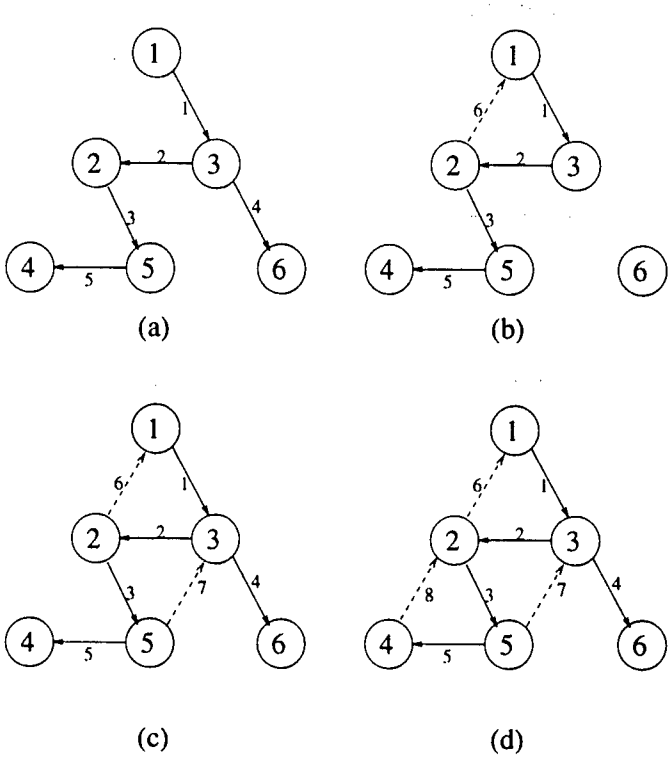


Figure 2.3: The four steps of Algorithm FFL which finds the four fundamental loops of the graph shown in Figure 2.1. For each iteration k, the subgraph $G_R^{(k)}$ is circled.

2.3 Scheduling and Retiming Formulations

Time scheduling (or simply scheduling) consists of assigning execution times to the operations in a DFG such that the precedence constraints of the DFG are not violated. This section considers two scheduling problems, namely, scheduling to a time-multiplexed bit-parallel target architecture (we call this *bit-parallel scheduling*) and scheduling to a bit-serial target architecture (we call this *bit-serial scheduling*). It turns out that the bit-parallel and bit-serial scheduling formulations are quite similar, and the retiming formulation is a special case of bit-parallel scheduling.

2.3.1 Bit-Parallel Scheduling

In bit-parallel scheduling, a DFG is statically scheduled to a bit-parallel target architecture. The scheduling formulation presented in this section is based on the folding equation developed in [28]. *Folding* is the process of executing several algorithm operations on a single hardware module. *Scheduling* is the process of determining at which time units a given algorithm operation is to be executed in hardware.

Before the scheduling formulation is developed, we need a brief description of retiming. The basic retiming equation for the edge $u \xrightarrow{e} v$ is [27]

$$w_r(e) = w(e) + r(v) - r(u), \tag{2.4}$$

where $w(e)$ is the number of delays on the edge before retiming, $w_r(e)$ is the number of delays on the edge after retiming, and $r(u)$ and $r(v)$ are the retiming values of nodes u and v, respectively.

The notions of an iteration and an iteration period are used in this section. An *iteration* is defined as the execution of each node in the DFG exactly once. The *iteration period* is defined as the number of clock cycles used to execute one iteration of the DFG in hardware.

Consider an edge e from node u to node v, denoted as $u \xrightarrow{e} v$. The operations (nodes) in the DFG are scheduled to be executed in the folded architecture once every N clock cycles, where N is the iteration period. Let the l-th iteration of nodes u and v be executed in hardware at time units $Nl + p(u)$ and $Nl + p(v)$, respectively, where $p(u)$ and $p(v)$ are the time partitions to which the nodes are scheduled to execute such that $0 \leq p(u), p(v) \leq N - 1$. Let edge e have $w_r(e)$ delays, which means that the result of the l-th iteration of node u is used by the $(l + w_r(e))$-th iteration of node v. The hardware modules which execute nodes u and v are denoted as H_u and H_v, respectively. If H_u is

pipelined by $d(u)$ stages, then the result of the l-th iteration of node u is available at $Nl + p(u) + d(u)$. This sample is used by the $(l + w_r(e))$-th iteration of node v, which is executed by H_v at $N(l + w_r(e)) + p(v)$, so the sample must be stored for

$$f(e) = N(l + w_r(e)) + p(v) - (Nl + p(u) + d(u)) = Nw_r(e) - d(u) + p(v) - p(u)$$

clock cycles. Substituting for $w_r(e)$ using (2.4) gives

$$f(e) = Nw(e) - d(u) - N(r(u) - r(v)) - (p(u) - p(v)). \tag{2.5}$$

The edge $u \xrightarrow{e} v$ with $w(e)$ delays in the DFG maps to an edge from H_u to H_v with $f(e)$ delays in the architecture, and the data on this edge are switched into H_v at time units $Nl + p(v)$.

Note that we assume that the hardware module H_u is pipelined by $d(u)$ delays, where $d(u)$ is the computation time of the node u in the DFG. If we define an $|E| \times 1$ vector $\mathbf{d}_u$ whose i-th element is the computation time of the source node of edge i (the source node of an edge is the node that the edge is incident from), then the folding equation can be written for all $|E|$ edges of the DFG simultaneously using

$$\mathbf{f} = N\mathbf{w} - \mathbf{d}_u - \mathbf{A}^T(\mathbf{p} + N\mathbf{r}), \tag{2.6}$$

where $\mathbf{A}$ is the $|V| \times |E|$ incidence matrix for the graph G (see Section 2.2.2), $\mathbf{p}$ is the $|V| \times 1$ *time partition vector* which assigns node i to the time partition p_i ($0 \leq p_i \leq N-1$), $\mathbf{r}$ is the $|V| \times 1$ *retiming vector* with the retiming values of the nodes in G, $\mathbf{w}$ is $|E| \times 1$ and contains the number of delays on each edge of G, $\mathbf{f}$ is the $|E| \times 1$ *folding vector* which contains the number of delays on each edge of the folded architecture, and $\mathbf{d}_u$ is the $|E| \times 1$ delay vector as previously described. This formulation of folding is general because it relies upon the retiming solution $\mathbf{r}$ and the time partition vector $\mathbf{p}$. One way to view this is that the DFG is preprocessed using retiming (hence the $\mathbf{r}$ vector) and

then scheduling is perfomed on the retimed DFG (hence the $\mathbf{p}$ vector). Combining $\mathbf{r}$ and $\mathbf{p}$ using $\mathbf{s} = \mathbf{p} + N\mathbf{r}$ results in the *schedule vector* $\mathbf{s}$. Using $\mathbf{s}$, the scheduling problem can be written as

$$\mathbf{A}^T\mathbf{s} = N\mathbf{w} - \mathbf{d}_u - \mathbf{f}. \tag{2.7}$$

The rank of the $|V| \times |E|$ incidence matrix $\mathbf{A}$ is $|V|-1$. Therefore, the left nullspace of $\mathbf{A}$ must consist of a vector $\mathbf{x}$ which satisfies $\mathbf{A}^T\mathbf{x} = \mathbf{0}_{|E|\times 1}$. We can see that $\mathbf{x} = \mathbf{1}_{|V|\times 1}$ because each column of $\mathbf{A}$ contains exactly one entry which is a 1, one entry which is a -1, and the remaining entries of the column are zero.

Using $\mathbf{A}^T\mathbf{1}_{|V|\times 1} = \mathbf{0}_{|E|\times 1}$ we can write

$$\mathbf{A}^T(\mathbf{s} + k\mathbf{1}) = N\mathbf{w} - \mathbf{d}_u - \mathbf{f},$$

which means that adding the constant k to each element of the schedule vector does not change the number of delays on the edges of the folded architecture.

The incidence matrix $\mathbf{A}$ can be written as

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_{|V|} \end{bmatrix}^T.$$

The reduced incidence matrix consists of any $|V| - 1$ rows of $\mathbf{A}$. Removing row m of $\mathbf{A}$ results in

$$\mathbf{A}_R = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_{m-1} & \mathbf{a}_{m+1} & \cdots & \mathbf{a}_{|V|} \end{bmatrix}^T. \tag{2.8}$$

The reduced incidence matrix $\mathbf{A}_R$ has dimensions $(|V| - 1) \times |E|$ and rank $|V| - 1$. The reduced scheduling vector is defined as

$$\mathbf{s}_R = \begin{bmatrix} s_1 & s_2 & \cdots & s_{m-1} & s_{m+1} & \cdots & s_{|V|} \end{bmatrix}^T, \tag{2.9}$$

which can be written as $\mathbf{s}_R = \mathbf{p}_R + N\mathbf{r}_R$, where $\mathbf{p}_R$ and $\mathbf{r}_R$ are the time partition vector $\mathbf{p}$ and the retiming vector $\mathbf{r}$ with the m-th elements removed. Using $\mathbf{A}_R$ and $\mathbf{s}_R$, we can

write

$$\mathbf{A}^T\mathbf{s} = s(m)\mathbf{a}_m + \mathbf{A}_R^T\mathbf{s}_R.$$

Substituting this into (2.7) results in

$$\mathbf{A}_R^T\mathbf{s}_R = N\mathbf{w} - \mathbf{d}_u - \mathbf{f} - s(m)\mathbf{a}_m. \tag{2.10}$$

Node m is called the *reference node.* Since replacing $\mathbf{s}$ by $\mathbf{s}' = \mathbf{s} + k\mathbf{1}$ does not alter the resulting folded architecture, we can choose $k = -s(m)$ so $s'(m) = 0$. After replacing $\mathbf{s}$ with $\mathbf{s}' = \mathbf{s} - s(m)\mathbf{1}$, (2.10) becomes $\mathbf{A}_R^T\mathbf{s}'_R = N\mathbf{w} - \mathbf{d}_u - \mathbf{f}$.

Throughout the remainder of this chapter, we will assume that $\mathbf{s}' = \mathbf{s} - s(m)\mathbf{1}$ so $s'(m) = 0$. In an abuse of notation, we will refer to $\mathbf{s}'$ simply as $\mathbf{s}$ so that (2.7) can be written as

$$\mathbf{A}_R^T\mathbf{s}_R = N\mathbf{w} - \mathbf{d}_u - \mathbf{f}. \tag{2.11}$$

Lemma 2.6 *The equation (2.11) can be solved for* $\mathbf{s}_R$ *if and only if* $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u) = \mathbf{B}\mathbf{f}$.

Proof: The equation (2.11) has a solution if and only if $N\mathbf{w} - \mathbf{d}_u - \mathbf{f}$ is in the $|V| - 1$ dimensional row space of $\mathbf{A}_R$. Equivalently, (2.11) has a solution if and only if $N\mathbf{w} - \mathbf{d}_u - \mathbf{f}$ is perpendicular to the $|E| - |V| + 1$ dimensional nullspace of $\mathbf{A}_R$ because the nullspace is the orthogonal complement of the row space in $\Re^{|E|}$. Since $\mathbf{B}\mathbf{A}_R^T = \mathbf{0}$ (see Section 2.2.2), the $|E| - |V| + 1$ rows of the fundamental loop matrix $\mathbf{B}$ form a basis for the nullspace of $\mathbf{A}_R$. Therefore, (2.11) has a solution if and only if $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u - \mathbf{f}) = \mathbf{0}$. □

To understand the meaning of $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u - \mathbf{f}) = \mathbf{0}$, we begin by writing $\mathbf{B}$ as

$$\mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_{|E|-|V|+1} \end{bmatrix}^T$$

such that $\mathbf{b}_i^T$ is the i-th row of $\mathbf{B}$. Using this, $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u - \mathbf{f}) = \mathbf{0}$ implies $\mathbf{b}_i^T\mathbf{f} = \mathbf{b}_i^T(N\mathbf{w} - \mathbf{d}_u)$. Recall that $b_{ij} = 1$ if edge j is in loop i and $b_{ij} = 0$ otherwise. Therefore,

$\mathbf{b}_i^T\mathbf{f}$ is the total number of folded delays on loop i, and $\mathbf{b}_i^T(N\mathbf{w} - \mathbf{d}_u)$ is a constant that depends on G. The equation $\mathbf{b}_i^T\mathbf{f} = \mathbf{b}_i^T(N\mathbf{w} - \mathbf{d}_u)$ states that the number of folded delays on loop i is the same for any legal folding vector $\mathbf{f}$, and $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u - \mathbf{f}) = \mathbf{0}$ implies that this is true for all $|E| - |V| + 1$ independent loops of G represented by the rows of $\mathbf{B}$. Furthermore, the sum of the number of folded delays for all edges and pipelining delays associated with all nodes of a loop is the product of the folding factor, N, and the number of loop delay elements, as noted in [28]. It can also be shown that this holds for the dependent loops of G, i.e., the number of folded delays on each loop of G that is not represented by a row of $\mathbf{B}$ is the same for any legal folding vector $\mathbf{f}$.

If $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u) = \mathbf{Bf}$ holds, (2.11) has exactly one solution for $\mathbf{s}_R$, which is given by

$$\mathbf{s}_R = (\mathbf{A}_R\mathbf{A}_R^T)^{-1}\mathbf{A}_R(N\mathbf{w} - \mathbf{d}_u - \mathbf{f}). \tag{2.12}$$

The above discussion can be summarized by saying that the number of folded delays on each loop in G is the same for any valid schedule $\mathbf{s}$.

In addition to the condition $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u) = \mathbf{Bf}$ there is also the practical condition that the number of delays on an edge in the folded architecture must be nonnegative. This condition can be written as $\mathbf{f} \geq \mathbf{0}$. The constraints for a valid schedule are

1. $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u) = \mathbf{Bf}$
2. $\mathbf{f} \geq \mathbf{0}$.

2.3.2 Retiming

Retiming is the process of moving delays around in a circuit without changing the functionality of the circuit [27]. A brief description of retiming is given at the beginning of Section 2.3.1. This section describes how retiming can be viewed as a special case of bit-parallel scheduling.

The folding equation for a graph G is given in (2.6). If each node in G represents a hardware operator, then all operations in the graph are executed in a single clock cycle resulting in an iteration period of $N = 1$. The elements of the time partition vector $\mathbf{p}$ are all zero because time partition zero is the only available partition. If we let $\mathbf{d}_u = \mathbf{0}$, i.e., we do not consider any internal pipelining of the operators, (2.6) becomes

$$\mathbf{f} = (1)\mathbf{w} - \mathbf{0} - \mathbf{A}^T(\mathbf{0} + 1\mathbf{r})$$

which simplifies to

$$\mathbf{f} = \mathbf{w} - \mathbf{A}^T\mathbf{r}. \tag{2.13}$$

Since $\mathbf{f}$ is the number of delays in the folded architecture, $\mathbf{f}$ is equivalent to $\mathbf{w}_r$ for $N = 1$, so (2.13) becomes

$$\mathbf{w}_r = \mathbf{w} - \mathbf{A}^T\mathbf{r}, \tag{2.14}$$

which is simply the matrix notation for writing (2.4) simultaneously for all edges of the graph. This demonstrates that retiming is simply scheduling when the iteration period is unity.

Using $\mathbf{A}^T\mathbf{1}_{|V|\times 1} = \mathbf{0}_{|E|\times 1}$, (2.14) can be written as

$$\mathbf{A}^T(\mathbf{r} + k\mathbf{1}) = \mathbf{w} - \mathbf{w}_r.$$

If $\mathbf{r}$ is a retiming vector which maps the graph G to the retimed graph G_r, then so is $(\mathbf{r} + k\mathbf{1})$ for any integer k.

In the context of retiming (i.e., assuming $N = 1$, $\mathbf{p} = \mathbf{0}$, $\mathbf{d}_u = \mathbf{0}$, and $\mathbf{f} = \mathbf{w}_r$), (2.11) can be written as

$$\mathbf{A}_R^T\mathbf{r}_R = \mathbf{w} - \mathbf{w}_r. \tag{2.15}$$

Recall that (2.11) assumes that $s(m) = 0$. Since $\mathbf{s} = N\mathbf{r} + \mathbf{p}$ and $\mathbf{p} = \mathbf{0}$ is assumed to obtain (2.15), this implies that $r(m) = 0$ in (2.15). In other words, the retiming value of the reference node is 0 in this formulation.

The translation of Lemma 2.6 to the retiming context is that (2.15) has a solution if and only if $\mathbf{Bw} = \mathbf{Bw}_r$ holds. This implies that the number of delays on any loop in G remains unchanged during retiming, as noted in [27]. If $\mathbf{Bw} = \mathbf{Bw}_r$ holds, (2.15) has exactly one solution for $\mathbf{r}_R$, which is given by

$$\mathbf{r}_R = (\mathbf{A}_R\mathbf{A}_R^T)^{-1}\mathbf{A}_R(\mathbf{w} - \mathbf{w}_r). \tag{2.16}$$

In addition to the condition $\mathbf{Bw} = \mathbf{Bw}_r$, there is also the practical condition that the number of delays on an edge in the retimed graph must be nonnegative. This condition can be written as $\mathbf{w}_r \geq \mathbf{0}$. The condition for a valid retiming from G to G_r are

1. $\mathbf{Bw} = \mathbf{Bw}_r$
2. $\mathbf{w}_r \geq \mathbf{0}$.

2.3.3 Bit-Serial Scheduling

In this section, a scheduling formulation is developed where the target architecture is a bit-serial architecture. This formulation, which is similar to the formulation in Chapter 6 of [50], has the same general form as the retiming and the bit-parallel scheduling formulations in Sections 2.3.1 and 2.3.2.

A bit-serial operator is often represented using a timing diagram such as the one in Figure 2.4. Let the execution of operator A in this figure begin at time T_A. The first bit of each of the inputs x_1, x_2, and x_3 arrives at time units $T_A + t(x_1)$, $T_A + t(x_2)$, and $T_A + t(x_3)$, respectively. The first bit of each of the outputs y_1 and y_2 is produced at time units $T_A + t(y_1)$ and $T_A + t(y_2)$, respectively. In other words, the timing diagram gives the relative differences between the timing of the input and output samples of the operator.

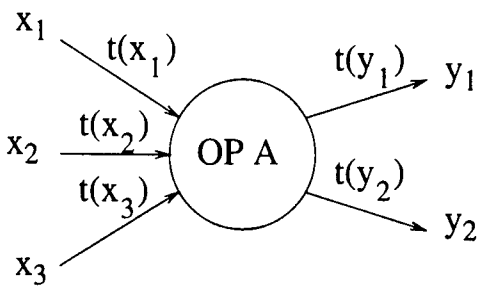
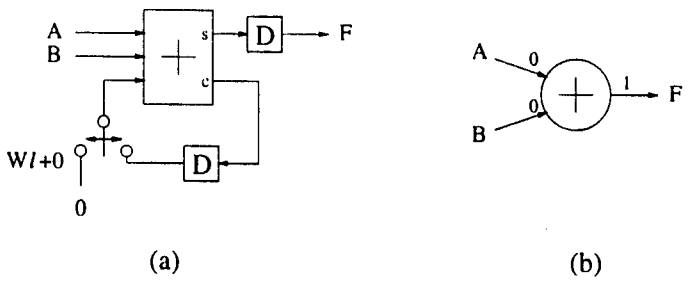


Figure 2.4: The timing diagram for the bit-serial operator A.

Example 2.3 *For the bit-serial adder in Figure 2.5(a) which computes $F = A + B$, the timing diagram is shown in Figure 2.5(b). Note that W is the wordlength.*



(a) (b)

Figure 2.5: (a) The architecture for a bit-serial adder for wordlength of W. (b) The timing diagram for this architecture.

The constraints for the bit-serial scheduling problem can be derived using the timing diagram. Consider the edge $u \xrightarrow{e} v$ with $w_r(e)$ delays in Figure 2.6. The output of iteration l of u is used as the input of iteration $l + w_r(e)$ of v. Let the l-th iteration of nodes u and v begin execution at time units $Wl + p(u)$ and $Wl + p(v)$, respectively, where W is the data wordlength and $p(u)$ and $p(v)$ are the time partitions to which the nodes are scheduled to execute such that $0 \leq p(u), p(v) \leq W - 1$. The output of the l-th iteration of u is available at $Wl + p(u) + t(u)$ and the output of the $l + w_r(e)$-th iteration of v is consumed at $W(l + w_r(e)) + p(v) + t(v)$, so the result must be stored for

$$b(e) = W(l+w_r(e))+p(v)+t(v)-[Wl+p(u)+t(u)] = Ww_r(e)-(t(u)-t(v))+p(v)-p(u)$$

clock cycles.

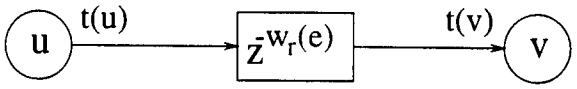


Figure 2.6: An edge $u \xrightarrow{e} v$ with $w_r(e)$ delays.

This equation can be written for all $|E|$ edges of the graph simultaneously according to

$$\mathbf{b} = W\mathbf{w}_r - (\mathbf{t}_u - \mathbf{t}_v) - \mathbf{A}^T\mathbf{p}, \tag{2.17}$$

where

- $\mathbf{A}$ is the incidence matrix for the graph.
- $\mathbf{p}$ is the time partition vector which assigns node i to the time partition p_i where $0 \leq p_i \leq W - 1$.
- $\mathbf{t}_u$ is defined such that t_{u_i} is the value $t(\cdot)$ at the source of edge i in the graph.
- $\mathbf{t}_v$ is defined such that t_{v_i} is the value $t(\cdot)$ at the sink of edge i in the graph.
- $\mathbf{w}_r$ contains the number of delays on each edge of the retimed DFG.
- $\mathbf{b}$ contains the number of *serial delays* on each edge of the hardware implementation.

The bit-serial folding equation (2.17) operates on the retimed DFG G_r. Substituting (2.14) into (2.17) results in

$$\mathbf{b} = W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v) - \mathbf{A}^T(\mathbf{p} + W\mathbf{r}).$$

Combining $\mathbf{r}$ and $\mathbf{p}$ using $\mathbf{s} = \mathbf{p} + W\mathbf{r}$ results in

$$\mathbf{A}^T\mathbf{s} = W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v) - \mathbf{b}.$$

This equation can be rewritten as

$$\mathbf{A}_R^T \mathbf{s}_R = W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v) - \mathbf{b}, \tag{2.18}$$

where $\mathbf{A}_R$ and $\mathbf{s}_R$ are defined as in (2.8) and (2.9), and the scheduling value for the reference node is $s(m) = 0$.

Using the same argument as in Lemma 2.6, it can be shown that the bit-serial scheduling equation (2.18) has a solution if and only if $\mathbf{B}(W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v)) = \mathbf{Bb}$. The equation $\mathbf{B}(W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v)) = \mathbf{Bb}$ states that the sum of the serial delays in any loop of the hardware implementation is the same for any valid serial delay vector $\mathbf{b}$. In addition, the sum of the number of serial delay elements of all edges and latencies associated with all nodes in a loop is the same as the product of the word-length and the number of loop delay elements.

A second constraint, $\mathbf{b} \geq \mathbf{0}$, exists because a connection in hardware cannot have a negative number of delays. The constraints for a valid bit-serial schedule are

1. $\mathbf{B}(W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v)) = \mathbf{Bb}$
2. $\mathbf{b} \geq \mathbf{0}$

The value of the schedule vector $\mathbf{s}$ can be found using

$$\mathbf{s}_R = (\mathbf{A}_R \mathbf{A}_R^T)^{-1} \mathbf{A}_R (W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v) - \mathbf{b}). \tag{2.19}$$

2.4 Generating All Scheduling and Retiming Solutions

2.4.1 Generating All Bit-Parallel Scheduling Solutions

Based on the two constraints $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u) = \mathbf{Bf}$ and $\mathbf{f} \geq \mathbf{0}$, all scheduling solutions for a strongly connected DFG can be generated. A systematic technique for generating these

solutions is presented in this section.

Recall that $\mathbf{B}$ is the fundamental loop matrix which can be expressed as $\mathbf{B} = \left[\begin{array}{c|c} \mathbf{C} & \mathbf{L} \end{array} \right]$, where $\mathbf{C}$ is an $(|E| - |V| + 1) \times (|V| - 1)$ matrix and $\mathbf{L}$ is an $(|E| - |V| + 1) \times |E| - |V| + 1)$ lower triangular matrix with ones on the diagonal. The columns of $\mathbf{C}$ correspond to the branches of the spanning tree of G which is chosen before Algorithm FFL is used to find $\mathbf{B}$, and the columns of $\mathbf{L}$ correspond to the links of G. The rows of $\mathbf{B}$ correspond to $(|E| - |V| + 1)$ linearly independent loops in G.

The algorithm for generating all scheduling solutions requires an interval to be written for the folded weight of each branch of G and an equality to be written for the folded weight of each link of G. The interval for the folded weight of a branch gives the range of possible values for the number of folded delays for this branch in the folded architecture. The equality for the folded weight of a link gives an expression for the number of delays for the link in the folded architecture. Using these intervals and equalities, code can be constructed to generate all possible scheduling solutions.

To determine these intervals and equalities, the elements of the fundamental loop matrix are examined one-by-one in a row-by-row manner, starting at the top-left of the matrix. Each time a "1" is encountered in the $\mathbf{C}$ submatrix of $\mathbf{B}$ such that this "1" is the first "1" encountered in its column, an interval is specified for this branch. This interval, which represents the range for the number of folded delays for the branch in the folded architecture, takes into account the intervals and equalities previously determined in the row-by-row scan of $\mathbf{B}$.

Assume that the first "1" in column n of $\mathbf{C}$ is in row m, i.e., $b_{mn} = 1$ and $b_{ln} = 0$ for all $l < m$. Let $\mathbf{b}_k^T$ denote any row of $\mathbf{B}$ such that $b_{kn} = 1$, i.e., loop(k) is a fundamental loop that contains the edge n. Since b_{mn} is the first "1" in column n, $m \leq k \leq |E| - |V| + 1$

must hold, i.e., b_{kn} is in row m or in a row which is below row m. From $\mathbf{Bf} = \mathbf{B}(N\mathbf{w}-\mathbf{d}_u)$, we get

$$\mathbf{b}_k^T\mathbf{f} = \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u) \Rightarrow \sum_{j\in E} b_{kj}f_j = \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u) \Rightarrow f_n + \sum_{j\in E-\{n\}} b_{kj}f_j = \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u). \tag{2.20}$$

Let D denote the set of edges encountered before reaching the element b_{mn} in the row-by-row scan of $\mathbf{B}$. Mathematically, D is the set of edges j such that there exists an element $b_{ij} = 1$ such that $j + (|E|-1)i < n + (|E|-1)m$. Using D, we can rewrite (2.20) as

$$f_n + \sum_{j\in D} b_{kj}f_j + \sum_{j\in E-D-\{n\}} b_{kj}f_j = \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u). \tag{2.21}$$

The intervals and equalities for the edges in the set $E - D - \{n\}$ have not yet been determined; however, we do know from $\mathbf{f} \geq \mathbf{0}$ that $\sum_{j\in E-D-\{n\}} b_{kj}f_j \geq 0$. Using this in (2.21) results in

$$f_n + \sum_{j\in D} b_{kj}f_j \leq \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u).$$

Using this along with $\mathbf{f} \geq \mathbf{0}$ specifies the interval for f_n

$$0 \leq f_n \leq \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u) - \sum_{j\in D} b_{kj}f_j, \tag{2.22}$$

which must hold for all k such that $b_{kn} = 1$.

Because the matrix $\mathbf{L}$ in $\mathbf{B} = \left[\begin{array}{c|c} \mathbf{C} & \mathbf{L} \end{array}\right]$ is lower triangular with ones on the diagonal, the diagonal element of row m, l_{mm}, is always the first "1" encountered in column m of $\mathbf{L}$ during the row-by-row scan of $\mathbf{B}$. In addition to using l_{mm} to denote this element, it can also be denoted as b_{mn} where $n = |V| - 1 + m$. When b_{mn} is encountered in the row-by-row scan of $\mathbf{B}$ such that $n = |V| - 1 + m$, an equality is written for f_n based on the equation $\mathbf{b}_m^T\mathbf{f} = \mathbf{b}_m^T(N\mathbf{w}-\mathbf{d}_u)$. This equality, which uses the fact that the intervals and equalities have already been determined for all edges in loop(m) except

edge n, is

$$f_n = \mathbf{b}_m^T(N\mathbf{w} - \mathbf{d}_u) - \sum_{j \in D} b_{mj} f_j. \tag{2.23}$$

To summarize the above discussion, the matrix $\mathbf{B}$ is scanned in a row-by-row manner starting with $b_{1,1}$. When $b_{mn} = 1$ is encountered, if b_{mn} is the first "1" in its column of $\mathbf{C}$, the interval in (2.22) is written for all k such that $b_{kn} = 1$. When $b_{mn} = 1$ is encountered where $n = |V| - 1 + m$, the equality in (2.23) is written.

The intervals for the $|V| - 1$ branches of G are denoted as $\mathcal{I}_j$ for $1 \leq j \leq |V| - 1$. An algorithm for writing these $|V| - 1$ intervals for the branches and the $|E| - |V| + 1$ equalities for the links is given below. At any point in this algorithm, D is the set of edges in G whose intervals or equalities have previously been determined.

Algorithm IE (Intervals and Equalities)

$D = \{\}$;
FOR ($m = 1$ TO $|E| - |V| + 1$)
{
  FOR ($n = 1$ TO $|E| - 1$)
  {
    IF ($b_{mn} = 1$ AND $b_{kn} = 0\ \forall k < m$)
    {
      IF ($1 \leq n \leq |V| - 1$)
      {
        $\mathcal{I}_n = [0, \min\{\sigma(m,n), \sigma(m+1,n), \sigma(|E| - |V| + 1, n)\}]$;
        $D \leftarrow D + \{n\}$;
      }
      ELSE
      {
        $f_n = \mathbf{b}_m^T(N\mathbf{w} - \mathbf{d}_u) - \sum_{j \in D} b_{mj} f_j$;
        $D \leftarrow D + \{n\}$;
      }
    }
  }
}

where

$$\sigma(k,n) = \begin{cases} \mathbf{b}_k^T(N\mathbf{w} - \mathbf{d}_u) - \sum_{j \in D} b_{kj} f_j & \text{if } b_k = 1 \\ \infty & \text{otherwise} \end{cases}.$$

From the intervals and equalities, code can be written to enumerate all possible scheduling solutions. The general structure of the code is:

1. Write FOR loops for the intervals and write assignment statements for the equalities in the same order that these intervals and equalities are generated in Algorithm IE.

2. Test the link weights for non-negativity. If the link weights pass this test, the edge weights represent a valid scheduling solution.

This technique generates all possible scheduling solutions because the FOR loop for branch m assigns f_m every integer value which is legal under the constraints $\mathbf{Bf} = \mathbf{B}(N\mathbf{w} - \mathbf{d}_u)$ and $\mathbf{f} \geq \mathbf{0}$, while taking into consideration the values of f_i which are already contained in a FOR loop or an assignment statement.

Example 2.4 *In this example, we find all scheduling solutions for the DFG in Figure 2.7 assuming an iteration period of* 4 *and assuming that the computation time for each node is unity.*

$$N\mathbf{w} - \mathbf{d}_u = \begin{bmatrix} -1 & 3 & -1 & -1 & -1 & -1 & -1 & 3 & 3 \end{bmatrix}^T$$

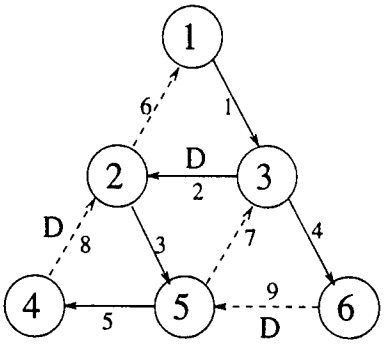


Figure 2.7: The data-flow graph used in Example 2.5.

and

$$\mathbf{Bf} = \mathbf{B}(N\mathbf{w} - \mathbf{d}_u) \Rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \\ f(9) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Using Algorithm IE gives the intervals and equalities

$$\begin{array}{cc} \mathcal{I}_1 = [0,1] & D = \{1\} \\ \mathcal{I}_2 = [0,1-f_1] & D = \{1,2\} \\ f_6 = 1 - f_1 - f_2 & D = \{1,2,6\} \\ \mathcal{I}_3 = [0,1-f_2] & D = \{1,2,3,6\} \\ f_7 = 1 - f_2 - f_3 & D = \{1,2,3,6,7\} \\ \mathcal{I}_5 = [0,1-f_3] & D = \{1,2,3,5,6,7\} \\ f_8 = 1 - f_3 - f_5 & D = \{1,2,3,5,6,7,8\} \\ \mathcal{I}_4 = [0,1-f_7] & D = \{1,2,3,4,5,6,7,8\} \\ f_9 = 1 - f_4 - f_7 & D = E. \end{array}$$

The code for finding all scheduling solutions is

```
for (f1 = 0; f1 <= 1; f1++)
for (f2 = 0; f2 <= 1 - f1; f2++)
{
    f6 = 1 - f1 - f2;
    for (f3 = 0; f3 <= 1 - f2; f3++)
    {
        f7 = 1 - f2 - f3;
```

Table 2.1: The twelve valid scheduling solutions for the DFG in Figure 2.7.

sol'n #	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	s_1	s_2	s_3	s_4	s_5	s_6
1	0	0	0	0	0	1	1	1	0	0	-2	1	0	-1	2
2	0	0	0	0	1	1	1	0	0	0	-2	1	1	-1	2
3	0	0	1	0	0	1	0	0	1	0	-2	1	1	0	2
4	0	0	1	1	0	1	0	0	0	0	-2	1	1	0	3
5	0	1	0	0	0	0	0	1	1	0	-1	1	1	0	2
6	0	1	0	1	0	0	0	1	0	0	-1	1	1	0	3
7	0	1	0	0	1	0	0	0	1	0	-1	1	2	0	2
8	0	1	0	1	1	0	0	0	0	0	-1	1	2	0	3
9	1	0	0	0	0	0	1	1	0	0	-1	2	1	0	3
10	1	0	0	0	1	0	1	0	0	0	-1	2	2	0	3
11	1	0	1	0	0	0	0	0	1	0	-1	2	2	1	3
12	1	0	1	1	0	0	0	0	0	0	-1	2	2	1	4

```
        for (f5 = 0; f5 <= 1 - f3; f5++)
        {
            f8 = 1 - f3 - f5;
            for (f4 = 0; f4 <= 1 - f7; f4++)
            {
                f9 = 1 - f4 - f7;
                if (f6 >= 0 AND f7 >= 0 AND f8 >= 0 AND f9 >= 0)
                   print the values of f1 through f9 and s1 through s6
            }
        }
    }
}
```

There are twelve scheduling solutions for this DFG. The scheduling vector $\mathbf{s}_R$ *can be computed from the folded edge vector* $\mathbf{f}$ *using (2.12). Using node 1 as the reference node, the folded edge weights and the scheduling values for the nodes are listed in Table 2.1.*

Once all possible $\mathbf{f}$ vectors have been found and the corresponding $\mathbf{s}$ vectors have been computed using (2.12), the $\mathbf{r}$ and $\mathbf{p}$ vectors can be found from $\mathbf{s}$ (recall that $\mathbf{s} = \mathbf{p} + N\mathbf{r}$) using $\mathbf{r} = \lfloor \frac{\mathbf{s}}{N} \rfloor$ and $\mathbf{p} = \mathbf{s} - N\mathbf{r}$. It can be shown that these expressions for $\mathbf{r}$ and $\mathbf{p}$ result in

- $0 \leq \mathbf{p} \leq N-1$. This means that p_i is indeed a time partition satisfying $0 \leq p_i \leq N-1$.

- $\mathbf{w}_r \geq \mathbf{0}$ and $\mathbf{Bw} = \mathbf{Bw}_r$. This means that $\mathbf{r}$ is a valid retiming solution of G.

To summarize, the following four steps can be used to find all valid schedules for a strongly connected DFG:

1. Find all vectors $\mathbf{f}$ such that $\mathbf{f} \geq 0$ and $\mathbf{Bf} = \mathbf{B}(N\mathbf{w} - \mathbf{d}_u)$.

2. Compute $\mathbf{s}$ using (2.12) and $s(m) = 0$, where m is the reference node.

3. $\mathbf{r} = \lfloor \frac{\mathbf{s}}{N} \rfloor$.

4. $\mathbf{p} = \mathbf{s} - N\mathbf{r}$.

These four steps give the valid schedules for G. The retiming vector $\mathbf{r}$ corresponds to a valid retiming solution for G, and the elements of the partition vector $\mathbf{p}$ satisfy $0 \leq p_i \leq N-1$.

For each legal folding vector $\mathbf{f}$, the technique in this section finds exactly one schedule $\mathbf{s}$, which contains information about the time partitions $\mathbf{p}$ and the retiming values $\mathbf{r}$ of the nodes. However, there are actually N schedules which map the DFG to a folded architecture which has $\mathbf{f}$ delays on its edges. We call these N solutions *equivalent schedules*, and we call the solution found using Step 2 above the *fundamental schedule* $\mathbf{s}$ of the folding vector $\mathbf{f}$. The N equivalent schedules are $\mathbf{s} + k\mathbf{1}$ for $0 \leq k \leq N-1$. Replacing $\mathbf{s}$ by $\mathbf{s} + k\mathbf{1}$ has two effects. First, the switching instance $Nl + j$ $(0 \leq j \leq N-1)$ in the folded architecture becomes $Nl + ((j+k) \mathrm{mod} N)$. Second, if scheduling is viewed as preprocessing the DFG by retiming (finding $\mathbf{r}$) and then assigning time partitions (finding $\mathbf{p}$), the preprocessed DFG may change because $\mathbf{r}$ may change. A nice property

of the technique presented in this section is that it finds the fundamental schedule $\mathbf{s}$ for each folding vector $\mathbf{f}$, and the N equivalent schedules are implicitly known to be $\mathbf{s} + k\mathbf{1}$ for $0 \leq k \leq N-1$.

2.4.2 Generating All Retiming Solutions

Since retiming is a special case of scheduling, the techniques in Section 2.4.1 for generating all scheduling solutions can also be used to generate all retiming solutions by replacing $\mathbf{f}$ with $\mathbf{w}_r$ and letting $N = 1$ and $\mathbf{d}_u = 0$.

Example 2.5 *In this example, we generate the edge intervals and equalities for the graph in Figure 2.7. The fundamental loop matrix for this graph is given in (2.1), the weight vector is*

$$\mathbf{w} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}^T,$$

and $\mathbf{Bw} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T$. *The intervals and equalities are generated in the following order using Algorithm IE.*

$$\begin{array}{cc} \mathcal{I}_1 = [0,1] & D = \{1\} \\ \mathcal{I}_2 = [0, 1 - w_{r_1}] & D = \{1,2\} \\ w_{r_6} = 1 - w_{r_1} - w_{r_2} & D = \{1,2,6\} \\ \mathcal{I}_3 = [0, 1 - w_{r_2}] & D = \{1,2,3,6\} \\ w_{r_7} = 1 - w_{r_2} - w_{r_3} & D = \{1,2,3,6,7\} \\ \mathcal{I}_5 = [0, 1 - w_{r_3}] & D = \{1,2,3,5,6,7\} \\ w_{r_8} = 1 - w_{r_3} - w_{r_5} & D = \{1,2,3,5,6,7,8\} \\ \mathcal{I}_4 = [0, 1 - w_{r_7}] & D = \{1,2,3,4,5,6,7,8\} \\ w_{r_9} = 1 - w_{r_4} - w_{r_7} & D = E \end{array}$$

Using these intervals and equalities, the code which generates all retiming solutions for the DFG in Figure 2.7 is given below. Note that xi *is used to represent* w_{r_i}.

```
for (x1 = 0; x1 <= 1; x1++)
for (x2 = 0; x2 <= 1 - x1; x2++)
{
    x6 = 1 - x1 - x2;
    for (x3 = 0; x3 <= 1 - x2; x3++)
```

```
    {
        x7 = 1 - x2 - x3;
        for (x5 = 0; x5 <= 1 - x3; x5++)
        {
            x8 = 1 - x3 - x5;
            for (x4 = 0; x4 <= 1 - x7; x4++)
            {
                x9 = 1 - x4 - x7;
                if (x6 >= 0 AND x7 >= 0 AND x8 >= 0 AND x9 >= 0)
                   print the values of x1 through x9 and r1 through r6
            }
        }
    }
}
```

There are twelve retiming solutions for the DFG. The retiming vector $\mathbf{r}$ *is computed from the retimed weight vector* $\mathbf{w}_r$ *using (2.16) and* $r(1) = 0$*, where node 1 is the reference node. The retimed edge weights and the retiming values for the nodes are listed in Table 2.2.*

If a DFG is not strongly connected, it is possible to add edges to the DFG to make it strongly connected so all retiming solutions can be generated. Consider the biquad filter in Figure 2.8(a). This graph is not strongly connected because, for example, there is no path from the output node to the input node. To make this graph strongly connected, it can be modified by adding an edge from the output node to the input node as shown in Figure 2.8(b). The modified graph has a new loop IN $\rightarrow$ OUT $\rightarrow$ IN which has one delay. This loop forces the latency of the DFG to be one cycle. Using the techniques presented in this section, we find that there are 224 retiming solutions for the DFG in Figure 2.8(b).

As another example, consider the correlator in Figure 2.9 which is used to demonstrate retiming in [27]. Using the techniques presented in this section, 143 retiming solutions can be found for this DFG. This result was also reported in [30].

Table 2.2: The twelve valid retiming solutions for the DFG in Figure 2.7.

sol'n #	w_{r_1}	w_{r_2}	w_{r_3}	w_{r_4}	w_{r_5}	w_{r_6}	w_{r_7}	w_{r_8}	w_{r_9}
1	0	0	0	0	0	1	1	1	0
2	0	0	0	0	1	1	1	0	0
3	0	0	1	0	0	1	0	0	1
4	0	0	1	1	0	1	0	0	0
5	0	1	0	0	0	0	0	1	1
6	0	1	0	1	0	0	0	1	0
7	0	1	0	0	1	0	0	0	1
8	0	1	0	1	1	0	0	0	0
9	1	0	0	0	0	0	1	1	0
10	1	0	0	0	1	0	1	0	0
11	1	0	1	0	0	0	0	0	1
12	1	0	1	1	0	0	0	0	0

sol'n #	r_1	r_2	r_3	r_4	r_5	r_6
1	0	-1	0	-1	-1	0
2	0	-1	0	0	-1	0
3	0	-1	0	0	0	0
4	0	-1	0	0	0	1
5	0	0	0	0	0	0
6	0	0	0	0	0	1
7	0	0	0	1	0	0
8	0	0	0	1	0	1
9	0	0	1	0	0	1
10	0	0	1	1	0	1
11	0	0	1	1	1	1
12	0	0	1	1	1	2

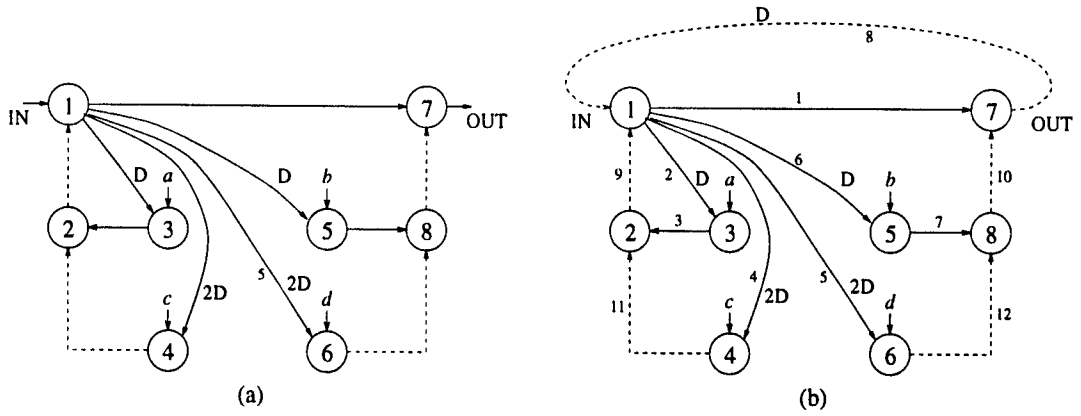


Figure 2.8: (a) The biquad filter. This graph is not strongly connected. (b) A modified version of the biquad filter. This graph is strongly connected.

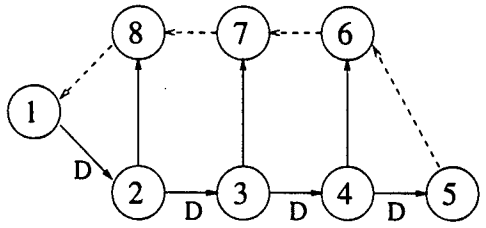


Figure 2.9: The correlator example which has 143 retiming solutions.

2.4.3 Bit-Serial Scheduling

Since the bit-serial scheduling formulation has the same form as the bit-parallel scheduling formulation, the techniques used to generate all bit-parallel scheduling solutions can be used to generate all bit-serial scheduling solutions by replacing $\mathbf{f}$ with $\mathbf{b}$ and replacing $N\mathbf{w} - \mathbf{d}_u$ with $W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v)$.

The values of $\mathbf{r}$ and $\mathbf{p}$ can be computed from $\mathbf{s}$ (recall that $\mathbf{s} = \mathbf{p} + W\mathbf{r}$) using $\mathbf{r} = \lfloor \frac{\mathbf{s}}{W} \rfloor$ and $\mathbf{p} = \mathbf{s} - W\mathbf{r}$. It can be shown that these expressions for $\mathbf{r}$ and $\mathbf{p}$ result in

- $0 \leq \mathbf{p} \leq N - 1$. This means that p_i is indeed a time partition satisfying $0 \leq p_i \leq N - 1$.

- $\mathbf{w}_r \geq \mathbf{0}$ and $\mathbf{Bw} = \mathbf{Bw}_r$ if $t_u \geq t_v$ for all edges $u \xrightarrow{e} v$ as shown in Figure 2.6. This

means that $\mathbf{r}$ is a valid retiming solution of G when $t_u \geq t_v$ for all $e \in E$.

Example 2.6 *In this example, we generate all possible schedules for the bit-serial implementation of the third-order all-pole filter shown in Figure 2.10 assuming two's complement number representation, data wordlength is 8 (i.e., $W = 8$), and coefficient wordlength is 4.*

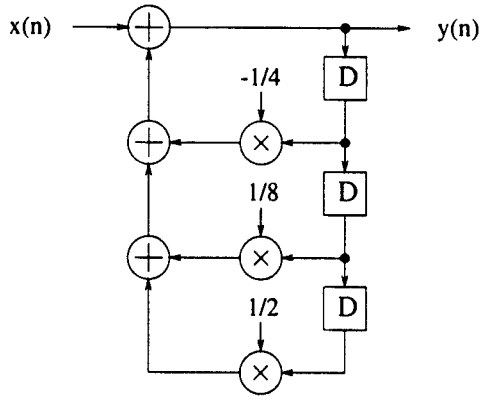


Figure 2.10: A third-order all-pole IIR filter.

The first step is to determine the timing diagram for each operator. The circuit and timing diagram for an adder are given in Figure 2.5. The circuits and timing diagrams for multiplication by $-1/4$, $1/8$, and $1/2$ are given in parts (a), (b), and (c), respectively, of Figure 2.11. Using these sub-circuits, the timing diagram for the filter is shown in Figure 2.12

The fundamental loop matrix is

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

In addition, we have

$$\mathbf{w} = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T,$$
$$\mathbf{t}_u = \begin{bmatrix} 1 & 1 & 1 & 4 & 3 & 1 & 1 & 1 \end{bmatrix}^T,$$

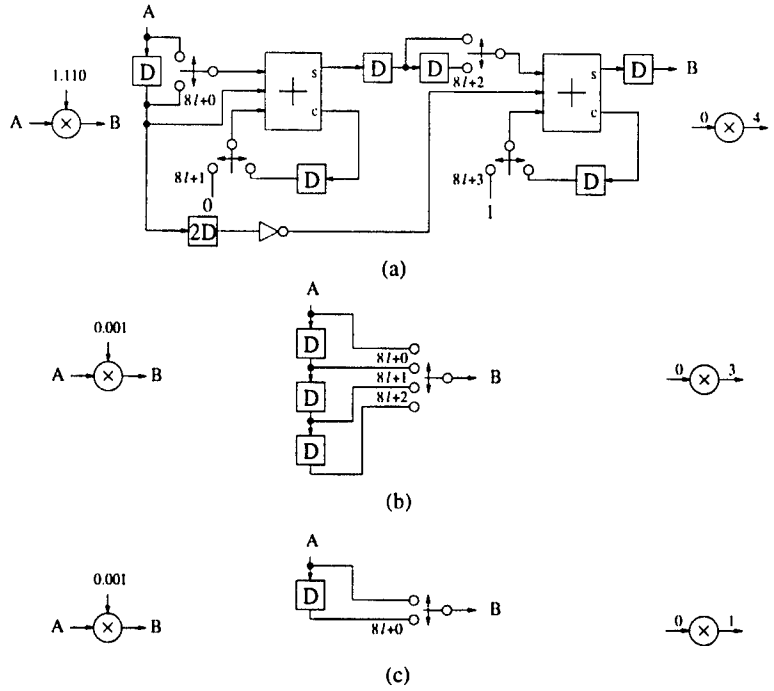


Figure 2.11: The circuits and timing diagrams for the three multipliers in Figure 2.10.

and $\mathbf{t}_v = \mathbf{0}$. *The equation* $\mathbf{B}(W\mathbf{w} - (\mathbf{t}_u - \mathbf{t}_v)) = \mathbf{B}\mathbf{b}$ *is*

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \mathbf{b} = \begin{bmatrix} 2 \\ 10 \\ 20 \end{bmatrix}.$$

The intervals and equalities are

$$\begin{aligned} \mathcal{I}_1 &= [0, 2] \\ \mathcal{I}_4 &= [0, 2 - b_1] \\ b_6 &= 2 - b_1 - b_4 \\ \mathcal{I}_2 &= [0, 10 - b_6] \\ \mathcal{I}_5 &= [0, 10 - b_2 - b_6] \\ b_7 &= 10 - b_2 - b_5 - b_6 \\ \mathcal{I}_3 &= [0, 20 - b_6 - b_7] \\ b_8 &= 20 - b_3 - b_6 - b_7 \end{aligned}$$

There are 6103 *valid scheduling solutions. To avoid examining all of these solutions, let us examine only those solutions which use the minimum number of serial registers. The number of serial registers is*

$$D = \max(b_1, b_2, b_3) + b_4 + b_5 + b_6 + b_7 + b_8.$$

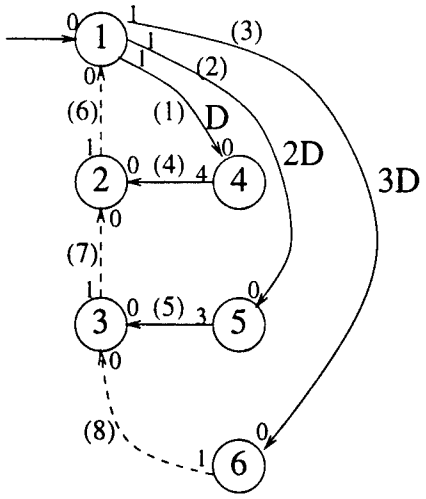


Figure 2.12: The timing diagram for the filter in Figure 2.10. The edge labels are shown in parentheses to avoid confusion with the timing values.

The minimum number of registers for all 6103 *valid scheduling solutions is* $D_{min} = 20$, *and there are* 330 *solutions which use* 20 *registers. One solution that uses* 20 *registers is*

$$\mathbf{b} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 2 & 8 & 10 \end{bmatrix}^T$$

$$\mathbf{s} = \begin{bmatrix} 0 & -3 & -12 & -7 & -15 & -23 \end{bmatrix}^T$$

$$\mathbf{r} = \begin{bmatrix} 0 & -1 & -2 & -1 & -2 & -3 \end{bmatrix}^T$$

$$\mathbf{p} = \begin{bmatrix} 0 & 5 & 4 & 1 & 1 & 1 \end{bmatrix}^T.$$

The complete architecture for this solution is shown in Figure 2.13. This architecture uses 20 registers, not including the registers which are internal to the processing units.

2.5 Bit-Parallel Scheduling with Resource Constraints

When all of the schedules are generated for a DFG, this may include many schedules which require more hardware resources than are available for the implementation. In this section, we describe two methods for finding the schedules which satisfy a given

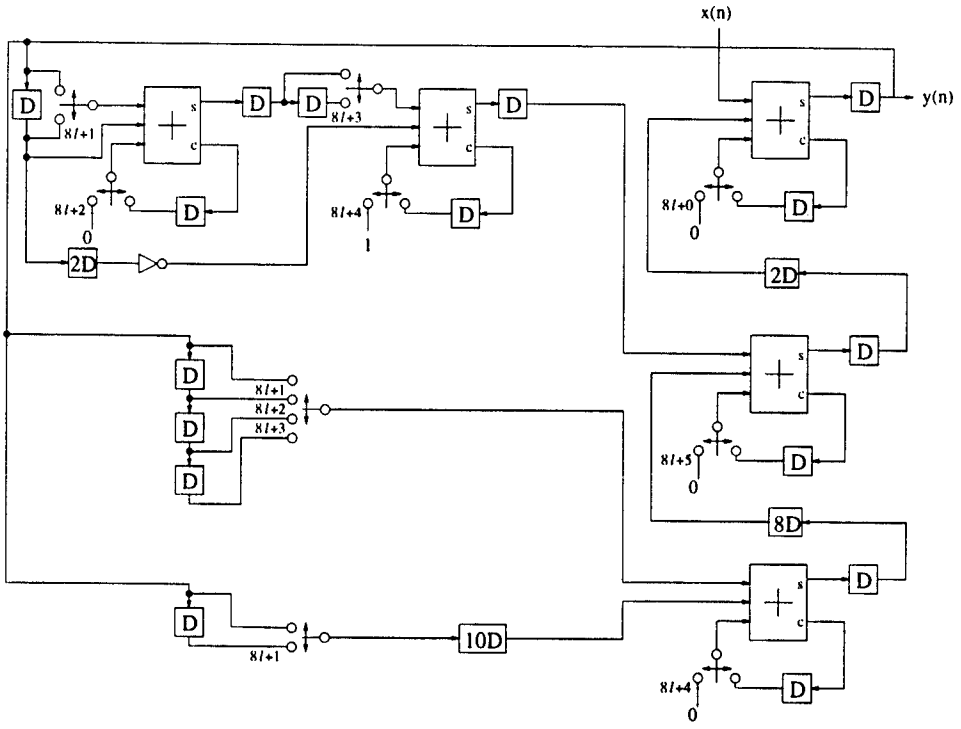


Figure 2.13: An architecture for the third-order all-pole filter. This architecture uses the minimum number of registers (20), not including the registers which are internal to the processing units.

set of resource constraints. In the first method (the *solution-save method*), we generate all scheduling solutions and then save only the solutions which satisfy the resource constraints. In the second method (the *solution-generate method*), we only generate those scheduling solutions which satisfy the resource constraints.

2.5.1 The Solution-Save Method

The number of hardware modules required by a scheduled DFG can be determined from **p**. For example, let m_n be the number of multiplication operations scheduled to time partition n ($0 \leq n \leq N-1$), and let a_n be the number of addition operations scheduled to time partition n. Then the number of multipliers required by the schedule is $m = \max_{0 \leq n \leq N-1}\{m_n\}$ and the number of adders is $a = \max_{0 \leq n \leq N-1}\{a_n\}$.

Example 2.7 *In this example we find all scheduling solutions which require 1 multiplier and 1 adder for the biquad filter in Figure 2.8(b) assuming an iteration period of* $N = 4$ *and assuming that addition and multiplication require 1 and 2 units of time, respectively. Nodes 1, 2, 7, and 8 are addition operations and nodes 3, 4, 5, and 6 are multiplication operations.*

The fundamental loop matrix is

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

and $\mathbf{B}(4\mathbf{w} - \mathbf{d}_u) = \begin{bmatrix} 2 & 0 & 3 & 4 & 7 \end{bmatrix}^T$. *The intervals and equalities are*

$$\begin{aligned} &\mathcal{I}_1 = [0, 2] \\ &f_8 = 2 - f_1 \\ &\mathcal{I}_2 = [0, 0] \Rightarrow f_2 = 0 \\ &\mathcal{I}_3 = [0, 0 - f_2] \Rightarrow f_3 = 0 \\ &f_9 = 0 - f_2 - f_3 \Rightarrow f_9 = 0 \\ &\mathcal{I}_6 = [0, 3 - f_8] \\ &\mathcal{I}_7 = [0, 3 - f_6 - f_8] \\ &f_{10} = 2 - f_6 - f_7 - f_8 \\ &\mathcal{I}_4 = [0, 4 - f_9] \\ &f_{11} = 4 - f_4 - f_9 \\ &\mathcal{I}_5 = [0, 7 - f_8 - f_{10}] \\ &f_{12} = 7 - f_5 - f_8 - f_{10} \end{aligned}$$

There is a total of 625 valid scheduling solutions for this example; however, only 6 of these solutions use only 1 adder and 1 multiplier. Tables 2.3 and 2.4 give the details of these solutions, and the DFGs for these six solutions are given in Figure 2.14.

Example 2.8 *Consider the 4-stage pipelined 8-th order all-pole lattice filter in Figure 2.15. Edge* 11 *has been added to this filter to make it strongly connected. For the iteration period* $N = 2$, *this filter has* 450 *scheduling solutions, and* 99 *of these schedules use 2 adders and 2 multipliers. Of these* 99 *schedules, the minimum possi-*

Table 2.3: The **f** and **s** values for the six valid scheduling solutions for the biquad filter which use 1 adder and 1 multiplier for an iteration period of 4.

sol'n #	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}
1	1	0	0	2	3	1	1	1	0	0	2	3
2	1	0	0	3	2	1	1	1	0	0	1	4
3	1	0	0	3	6	1	1	1	0	0	1	0
4	1	0	0	1	3	2	0	1	0	0	3	3
5	1	0	0	3	1	2	0	1	0	0	1	5
6	1	0	0	3	5	2	0	1	0	0	1	1

sol'n #	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
1	0	-1	-3	-5	-2	-4	2	1
2	0	-1	-3	-4	-2	-5	2	1
3	0	-1	-3	-4	-2	-1	2	1
4	0	-1	-3	-6	-1	-4	2	1
5	0	-1	-3	-4	-1	-6	2	1
6	0	-1	-3	-4	-1	-2	2	1

Table 2.4: The **r** and **p** values for the six valid scheduling solutions for the biquad filter which use 1 adder and 1 multiplier for an iteration period of 4.

sol'n #	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	0	-1	-1	-2	-1	-1	0	0	0	3	1	3	2	0	2	1
2	0	-1	-1	-1	-1	-2	0	0	0	3	1	0	2	3	2	1
3	0	-1	-1	-1	-1	-1	0	0	0	3	1	0	2	3	2	1
4	0	-1	-1	-2	-1	-1	0	0	0	3	1	2	3	0	2	1
5	0	-1	-1	-1	-1	-2	0	0	0	3	1	0	3	2	2	1
6	0	-1	-1	-1	-1	-1	0	0	0	3	1	0	3	2	2	1

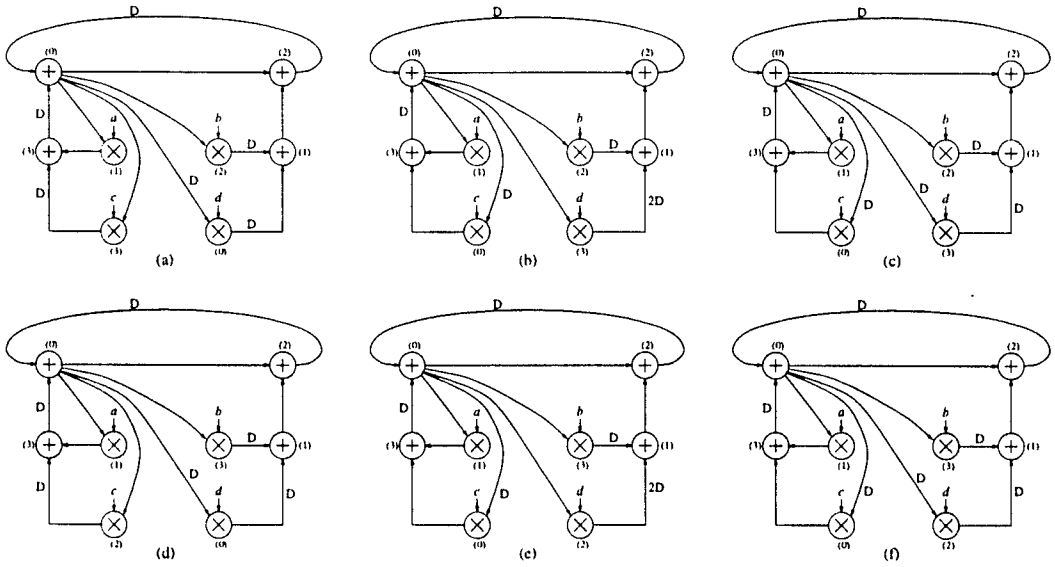


Figure 2.14: The six scheduling solutions for the biquad filter which use 1 adder and 1 multiplier. The number in parentheses next to a node is the time partition to which the node is scheduled.

ble number of registers required for the implementation is 10, *and only* 2 *of these* 99 *schedules use* 10 *registers. These schedules are* $\mathbf{s} = \begin{bmatrix} 0 & 3 & 1 & -2 & 1 & 4 & 2 & -1 \end{bmatrix}^T$ *and* $\mathbf{s} = \begin{bmatrix} 0 & 3 & 1 & -2 & 2 & 5 & 3 & 0 \end{bmatrix}^T$. *The minimum number of registers is computed using the techniques in [46] with the modification that the results reported here assume that for a processor that is pipelined by* P_u *stages, the* P_u *pipelining registers cannot be used by output samples from other processors, while the results in [46] allow one pipelining register to be shared by other processors. For the iteration period* $N = 4$, *the filter in Figure 2.15 has* 910910 *scheduling solutions, and* 10083 *of these schedules use 1 adder and 1 multiplier. Of these* 10083 *schedules, the minimum possible number of registers required for the implementation is* 11, *and* 21 *of these* 10083 *solutions use* 11 *registers.*

2.5.2 The Solution-Generate Method

This section describes a technique for exhaustively generating only the bit-parallel schedules which can be implemented on a given set of hardware resources. Using this tech-

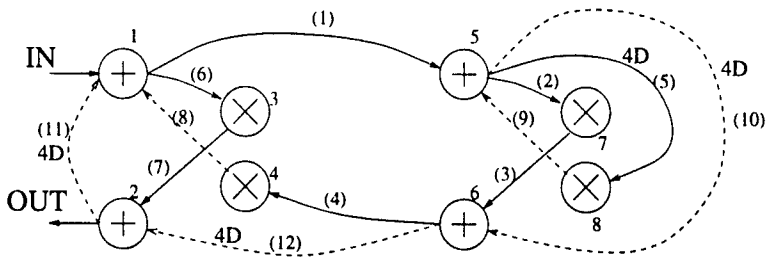


Figure 2.15: The 4-stage pipelined 8-th order all-pole lattice filter. The edge labels are in parentheses to avoid confusion with the node labels. One possible spanning tree is shown in solid lines.

nique, we can avoid generating those schedules which use more resources than are available, and this allows us to generate the desirable schedules in considerably less time. The following theorem is needed so we can construct **B** in a manner that allows us to perform exhaustive bit-parallel scheduling with resource constraints.

Theorem 2.7 *In Algorithm FFL, let v_J be the node that the link l_k is incident from. If v_J is in $G_R^{(k)}$, then there are no branches in loop(k) which are also in $(G-G_R^{(k)})$. If v_J is in $(G-G_R^{(k)})$, then there are branches in loop(k) which are in $(G-G_R^{(k)})$, and these branches form an elementary directed path which we shall denote as $v_0 \stackrel{b_1}{\rightarrow} v_1 \stackrel{b_2}{\rightarrow} \cdots \stackrel{b_{J-1}}{\rightarrow} v_{J-1} \stackrel{b_J}{\rightarrow} v_J$.*

Proof: The loop denoted as loop(k) in Algorithm FFL has the form of Figure 2.16(a) or 2.16(b), where v_R is the root node of the spanning tree and v_{IN} is a node in $G_R^{(k)}$. Recall from Theorem 2.5 that the form in Figure 2.16(b) results from $v_J \stackrel{l_k}{\rightarrow} v_{IN} \rightsquigarrow v_{COMMON} \rightsquigarrow v_R \rightsquigarrow v_{COMMON} \rightsquigarrow v_J$. Both forms of loop($k$) can be generalized as the loop in Figure 2.16(c), where v_{IN}, V_Y, and p_B are in $G_R^{(k)}$. The proof has two cases, which take into account whether or not node v_J is in $G_R^{(k)}$.

Case I: v_J is in $G_R^{(k)}$. If the path p_A in Figure 2.16(c) has any edges in $(G-G_R^{(k)})$, then a subpath $v_1 \rightsquigarrow v_2$ of p_A must exist in $(G-G_R^{(k)})$, where v_2 is in $G_R^{(k)}$. The last edge in $v_1 \rightsquigarrow v_2$, i.e., the edge that is incident into v_2, cannot be a link because l_k is the only

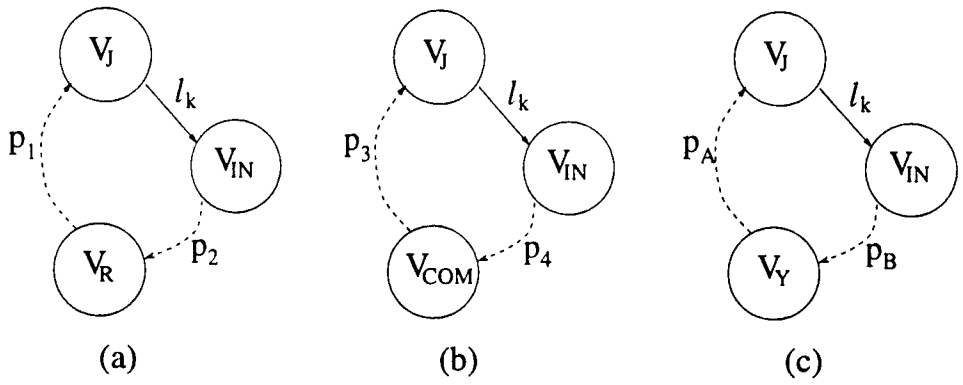


Figure 2.16: (a) One form of loop(k): $v_J \xrightarrow{l_k} v_{IN} \rightsquigarrow v_R \rightsquigarrow v_J$. Link l_k is in $(G - G_R^{(k)})$ and path p_2 is in $G_R^{(k)}$. (b) The other form of loop(k): $v_J \xrightarrow{l_k} v_{IN} \rightsquigarrow v_{COMMON} \rightsquigarrow v_J$. Link l_k is in $(G - G_R^{(k)})$ and path p_4 is in $G_R^{(k)}$. (c) Equivalent loop(k): $v_J \xrightarrow{l_k} v_{IN} \rightsquigarrow v_Y \rightsquigarrow v_J$. Link l_k is in $(G - G_R^{(k)})$ and path p_B is in $G_R^{(k)}$. The forms in (a) and (b) can be generalized to the form in (c).

link which is in loop(k) and in $(G - G_R^{(k)})$ (recall that loop(k) is in $G_R^{(k)} \cup G_T \cup l_k$). The last edge in $v_1 \rightsquigarrow v_2$ cannot be a branch because Lemma 2.3 says there is no branch in $(G - G_R^{(k)})$ which is incident into a node in $G_R^{(k)}$. Therefore, if v_J is in $G_R^{(k)}$, p_A can have no edges in $(G - G_R^{(k)})$, and there are no branches that are in loop(k) and in $(G - G_R^{(k)})$.

Case II: v_J is in $(G - G_R^{(k)})$. The edge incident into v_J in loop(k) is in $(G - G_R^{(k)})$ (if not, v_J would be in $G_R^{(k)}$), and this edge is a branch because l_k is the only link which is in loop(k) and in $(G - G_R^{(k)})$. We denote the branch in loop(k) which is incident into v_J as $v_{J-1} \xrightarrow{b_J} v_J$. Similarly, if v_{J-1} is in $(G - G_R^{(k)})$, then branch b_{J-1} exists in $(G - G_R^{(k)})$ to form the path $v_{J-2} \xrightarrow{b_{J-1}} v_{J-1} \xrightarrow{b_J} v_J$. On the other hand, if v_{J-1} is in $G_R^{(k)}$, then by using Case I of this proof, we know that the path $v_Y \rightsquigarrow v_{J-1}$ can have no edges in $(G - G_R^{(k)})$. Continuing this argument, we see that when v_J is in $(G - G_R^{(k)})$, there are branches which are in loop(k) and in $(G - G_R^{(k)})$, and these branches form the path $v_0 \xrightarrow{b_1} v_1 \xrightarrow{b_2} \cdots \xrightarrow{b_{J-1}} v_{J-1} \xrightarrow{b_J} v_J$. □

As described in Section 2.2.3, we construct the fundamental loop matrix $\mathbf{B}$ by letting loop(k) from Algorithm FFL be the k-th row of $\mathbf{B}$. The edges in the graph are numbered such that the first $(|V|-1)$ columns of $\mathbf{B}$ correspond to the branches of the spanning tree of G, and the remaining $(|E|-|V|+1)$ columns correspond to the links. From Theorem 2.7 we know that if there are branches in loop(k) which are in $(G-G_R^{(k)})$, then these branches form the elementary directed path $v_0 \stackrel{b_1}{\rightarrow} v_1 \stackrel{b_2}{\rightarrow} \cdots \stackrel{b_{J-1}}{\rightarrow} v_{J-1} \stackrel{b_J}{\rightarrow} v_J$. In other words, if loop(k) contains branches which have not appeared in previous loops, then these branches form a path. These branches are assigned to the next available columns of $\mathbf{B}$ in the order that they appear in the path $v_0 \stackrel{b_1}{\rightarrow} v_1 \stackrel{b_2}{\rightarrow} \cdots \stackrel{b_{J-1}}{\rightarrow} v_{J-1} \stackrel{b_J}{\rightarrow} v_J$. The link l_k is assigned to the $(|V|-1+k)$-th column of $\mathbf{B}$. By constructing the fundamental loop matrix in this manner, it still has the form given in (2.3); however, it now allows us to use Algorithm IE to determine the schedule values of the nodes directly.

The interval $\mathcal{I}_n$ for the scheduling problem is found by enforcing (2.22) for all k such that $b_{kn}=1$. Assume that the edge n is incident into node v_n and incident from node u_n, i.e., $u_n \stackrel{n}{\rightarrow} v_n$. From (2.7), the expression for the n-th folded edge weight is $f_n = Nw_n - d_{u_n} + s_{v_n} - s_{u_n}$. Substituting this into the interval for f_n gives

$$0 \leq Nw_n - d_{u_n} + s_{v_n} - s_{u_n} \leq \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u) - \sum_{j\in D} b_{kj}f_j$$

for all k such that $b_{kn}=1$. Solving for s_{v_n} gives

$$-Nw_n + d_{u_n} + s_{u_n} \leq s_{v_n} \leq -Nw_n + d_{u_n} + s_{u_n} + \mathbf{b}_k^T(N\mathbf{w}-\mathbf{d}_u) - \sum_{j\in D} b_{kj}f_j$$

for all k such that $b_{kn}=1$.

To avoid confusion with the interval for f_n (recall that we denoted this as $\mathcal{I}_n$), the interval for s_{v_n} is denoted as $\mathcal{I}_n^v$. This notation specifies that $\mathcal{I}_n^v$ is an interval for the scheduling value of the node that edge n is incident into. Let $\alpha_n = -Nw_n + d_{u_n} + s_{u_n}$.

Then the interval $\mathcal{I}_n^v$ is simply the interval $\mathcal{I}_n$ from Algorithm IE with α_n added to the lower and upper bounds. We shall denote this as $\mathcal{I}_n^v = \mathcal{I}_n + \alpha_n$.

Using the technique described in this section for constructing the fundamental loop matrix $\mathbf{B}$, Algorithm IE can be used to determine the intervals $\mathcal{I}_n$ for the folded edge weights, and the intervals for the scheduling values for the nodes can be found using $\mathcal{I}_n^v = \mathcal{I}_n + \alpha_n$.

Example 2.9 *In this example, all possible scheduling solutions are generated for the DFG in Figure 2.17 for an iteration period of* 4 *by generating the solutions for* $\mathbf{s}$ *directly. The computation time for each node is assumed to be unity. Using the technique described in this section for constructing* $\mathbf{B}$ *results in*

$$\mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}. \tag{2.24}$$

Notice that the edge labels in Figure 2.17 are different than those used in Figure 2.7. The labels have been changed so the column numbers of $\mathbf{B}$ *in (2.24) correspond to the edge labels in Figure 2.17. Using* $\mathbf{B}(N\mathbf{w} - \mathbf{d}_u) = \mathbf{1}_{4\times 1}$, *the intervals are given in Table 2.5. Note that in this table* $f_n = Nw_n - d_{u_n} + s_{v_n} - s_{u_n}$ *has been used to simplify the upper bounds of the* $\mathcal{I}_n^v$ *intervals.*

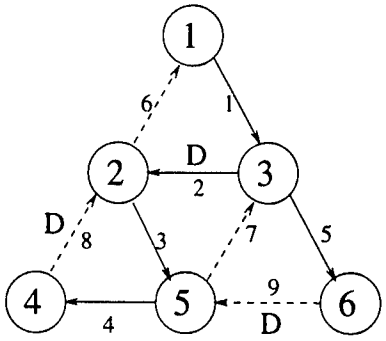


Figure 2.17: The graph scheduled in Example 2.9.

Table 2.5: The intervals for Example 2.9.

n	$\mathcal{I}_n$	α_n	$\mathcal{I}_n^v$
1	$[0, 1]$	$1 + s_1$	$[1, 2]$
2	$[0, 1 - f_1]$	$-3 + s_3$	$[-3 + s_3, -1]$
3	$[0, 1 - f_2]$	$1 + s_2$	$[1 + s_2, -1 + s_3]$
4	$[0, 1 - f_3]$	$1 + s_5$	$[1 + s_5, 3 + s_2]$
5	$[0, 1 - f_7]$	$1 + s_3$	$[1 + s_3, 3 + s_5]$

The code for this example is

```
for (s3 = 1; s3 <= 2; s3++)
for (s2 = -3 + s3; s2 <= -1; s2++)
for (s5 = 1 + s2; s5 <= -1 + s3; s5++)
for (s4 = 1 + s5; s4 <= 3 + s2; s4++)
for (s6 = 1 + s3; s6 <= 3 + s5; s6++)
{
    Compute link weights.  If all positive, print s1 through s6
}
```

The twelve solutions for s *generated from this code are the same as those listed in Table 2.1.*

By determining the values of the schedule vector directly rather than first determining the folding vector and then computing the schedule vector, we can generate only those schedules which can be executed using a limited number of hardware modules. This is done using a programming technique that avoids the solutions which use more resources than are available. For each operation type (e.g., addition or multiplication), an array of N data elements is used such that there is one element for each time partition from 0 to $N - 1$. Each data element contains the number of operations of a given type that is currently scheduled to that time partition. Each data element also keeps track of the next time partition in which the hardware resources for that particular operation

type are not fully utilized. By keeping track of this information, when we generate a new schedule by incrementing the schedule value for a node, the node is scheduled to a time partition in which the hardware resources for the operation are not already fully utilized. The end result is that we do not generate the schedules that use more resources than are available, so we can generate all scheduling solutions for a given set of resource constraints much more quickly than if we find all possible schedules and keep only those schedules which satisfy the resource constraints.

The advantages of including the resource constraints are demonstrated using the fifth-order wave digital elliptic filter shown in Figure 2.18. We assume that addition

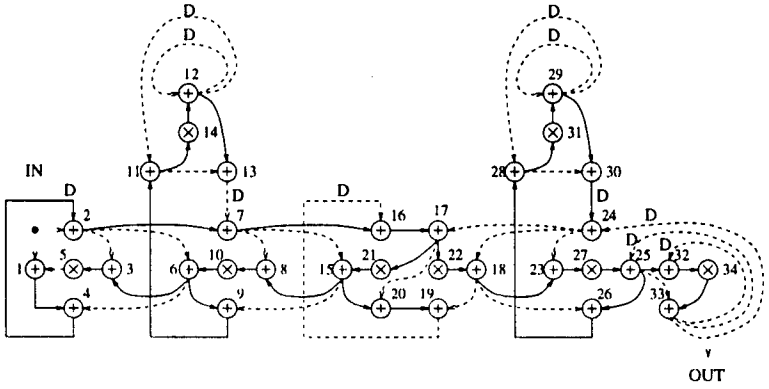


Figure 2.18: The fifth-order wave digital elliptic filter. The branches of the spanning tree used in Algorithm FFL is shown with solid lines, and the links are shown with dotted lines.

and multiplication require 1 and 2 units of time, respectively, and that hardware adders and multipliers are pipelined by 1 and 2 stages, respectively. The results of exhaustively generating the scheduling solutions without considering resource constraints are shown in Table 2.6. The results of exhaustively generating the scheduling solutions which can be implemented on a given number of hardware adders and multipliers are shown on the left side of Table 2.7. From these tables, we can see that the time it takes to exhaustively generate only the scheduling solutions which satisfy a given set of resource

Table 2.6: The results of exhaustively scheduling the filter in Figure 2.18 using the techniques presented in Section 2.4.1.

iter period	# sched solutions	CPU time (sec)
16	9900	0.0342
17	4669095	16.2
18	580432280	2020

Table 2.7: The results of exhaustively scheduling the filter in Figure 2.18 for a given set of resource constraints using the techniques presented in Section 2.5.2. The left part of the table considers scheduling to the minimum possible number of adders and multipliers for the given iteration period, and the right part considers scheduling to the minimum number of adders, multipliers, and registers.

iter period	resources (add,mult)	# solns	CPU time (sec)	resources (add,mult,reg)	# solns
16	(3, 1)	77	0.00288	(3, 1, 7)	21
17	(2, 1)	98	0.0518	(2, 1, 7)	73
18	(2, 1)	131983	11.1	(2, 1, 7)	40723
19	(2, 1)	33948842	1700	(2, 1, 7)	3056246

constraints is orders of magnitude faster than the time it takes to exhaustively generate all scheduling solutions. The expressions in [46] can be used to compute the number of registers required by a given schedule. The results of this are shown on the right side of Table 2.7. Note that these results assume that internal pipelining registers cannot be shared between processors, while the results in [46] assume that internal pipelining registers can be shared between processors.

2.6 Conclusions

Formulations have been presented in this chapter for the bit-parallel and bit-serial scheduling problems, and we have shown that the retiming formulation introduced in [30] is a special case of our bit-parallel scheduling formulation. Techniques have been developed and demonstrated for exhaustively generating all unique retiming and schedul-

ing solutions for a strongly connected DFG. These techniques allow a circuit designer to explore the space of possible implementations.

In addition to the technique for exhaustively generating all unique bit-parallel scheduling solutions, a technique was also developed for exhaustively generating only the bit-parallel scheduling solutions which satisfy a given set of resource constraints. Our results indicate that this technique can generate schedules in CPU times that are greater than two orders of magnitude faster than generating all solutions.

One advantage of the formulations presented in this chapter is that they allow us to understand how retiming and scheduling are similar and that retiming is an important part of scheduling. Specifically, we show that retiming is a special case of scheduling, and we include retiming in our scheduling formulations to make them general and to make visible the role of retiming during scheduling.

The numbers reported in Tables 2.6 and 2.7 show some scheduling results for the fifth-order wave digital elliptic filter. Since this filter is often used to demonstrate scheduling techniques, the numbers in these tables provide some benchmarks for gauging the effectiveness of scheduling algorithms. These numbers indicate that the number of schedules increases dramatically as the difference between the iteration period and the iteration bound becomes larger. Therefore, for practical applications, our exhaustive scheduling techniques are most useful when the iteration period is at or near the iteration bound.

Chapter 3

Register Minimization in Folded Architectures

3.1 Introduction

In this chapter, expressions are derived for the minimum number of registers required to implement a statically scheduled DFG. Two cases are considered, namely, the cases where retiming is and is not allowed to be perfomed on the scheduled DFG.

We begin with a motivating example. After the DFG has been scheduled, specifications for the communication paths between hardware modules can be determined using systematic folding techniques [28]. Consider the multiply-add operation in Figure 3.1(a), which is an algorithm DFG describing $y(n) = au(n) + v(n)$. Assume this multiply-add is part of a larger DFG which is to be implemented in hardware with an iteration period of 10, i.e., each node in the algorithm DFG will be executed by the hardware exactly once every 10 time units. If the multiply operation is executed by one-stage pipelined hardware module H_M at time units $10l + 2$, and the add operation is executed by hardware module H_A at $10l + 8$ for integer l iterations, then the connection between the multiplication and addition operations in Figure 3.1(a) is mapped to the data path in Figure 3.1(b) (details of how this data path specification is derived are provided in Sec-

tion 3.2.2). Upon examination of Figure 3.1(b), one observes that at any given time, no more than one of the five delays labeled "5D" between H_M and H_A is storing a word of data that will actually be consumed by H_A. To avoid the inefficient architecture that would result from direct implementation of Figure 3.1(b) in silicon, memory management is used in high-level synthesis tools to derive efficient data paths between processing modules.

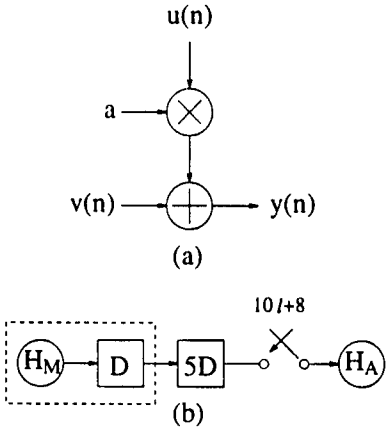


Figure 3.1: (a) Algorithm DFG describing $y(n) = au(n) + v(n)$. (b) Data path specification derived from the algorithm DFG for an iteration period of 10.

Memory management consists of choosing the type of registers, number of registers, and allocation of data to these registers. The type of registers is usually dictated by the architecture model used. Throughout this chapter, the term "register" is used to describe a storage location capable of storing one word of data. We use the term "memory model" for a general rule which describes how data can be allocated to the registers. For example, one memory model might force each functional unit in the architecture to store its output samples in a set of registers dedicated to only that functional unit, while another memory model might lift this restriction and allow all of the functional units to share a common set of registers. Naturally, the memory model affects the number of registers and the allocation of data to the registers. In this chapter, we compute

the minimum number of registers required for a statically scheduled DFG under various memory models. The allocation of the data to registers is an NP-complete problem for which heuristic algorithms have been suggested [51, 52, 53].

Techniques for computing the minimum number of registers required by a statically scheduled DFG have been considered in the past. The left-edge algorithm has been used to find the minimum number of registers and allocate data to these registers [54]. The life-time chart and circular life-time graph can be used to determine the minimum number of registers in any DSP circuit [29]. The circular life-time graph is particularly useful because it graphically takes into account the repetitive and periodic nature of DSP operations. These graphs have been used, for example, to determine the size of register files in DSP architectures [52].

In this chapter, we use life-time analysis to derive closed-form expressions for the minimum number of registers required by a statically scheduled DSP program. These techniques offer several advantages over previously used techniques. First, the closed-form expressions can be used to represent cost functions for high-level synthesis optimization tools. An example of using these closed-form expressions in an integer linear programming (ILP) formulation is given in Section 3.4. Second, the analytical tools we introduce can be used to derive expressions for the minimum number of registers under a variety of memory models which describe how data can be allocated to memory. This is important because the target architecture may impose constraints on how data can be routed to memory. We derive expressions for three memory models, namely the *operation-constrained*, *processor-constrained*, and *unconstrained* memory models. For the unconstrained memory model, where all memory-sharing constraints are relaxed, the minimum number of registers required to implement a DFG with m nodes can be computed in $O(m^2)$ time. A third advantage of the analytical tools we introduce is

that they can be used to determine memory requirements for more complex algorithm descriptions, such as DFGs which have multiplexers in the data paths.

Pipelining and retiming [27] are powerful tools used in high-level synthesis. Pipelining can be considered to be a special case of retiming. We consider an integer linear programming solution to the retiming problem, referred to as the minimum physical storage location (MPSL) retiming, which retimes a scheduled DFG such that its memory requirements are minimized under the unconstrained memory model while the schedule remains valid for the retimed DFG. We use MPSL retiming to retime a DFG which has been scheduled using the MARS design system [26], and we compare the memory requirements of MARS to a globally optimal solution. Our results show that the MARS system gives optimal or close-to-optimal results in terms of memory requirements.

The results we present can be used throughout the high-level synthesis process. Expressions for the minimum number of registers can be used during scheduling to help determine the total cost of the architecture. After scheduling, MPSL retiming can be used to optimally retime a DFG in terms of registers required for its implementation. During memory management, our techniques can be used to optimize the hardware design in terms of the number of registers required. For instance, given the scheduled DFG and the desired memory model, the minimum number of registers required can be determined, and register allocation can be performed by an appropriate register allocation scheme which guarantees completion (e.g., forward-backward register allocation [51]). Expressions for the minimum number of registers can also be used to evaluate the effectiveness of register allocation schemes which are based on heuristics, since some schemes may require more memory than the theoretical lower bound in order to maintain simple control structures.

This chapter is organized as follows. The algorithm DFG model and the pipelined pro-

cessor model used in the chapter are described in Section 3.2. This section also describes the systematic folding techniques which are used as a framework for our derivations. Expressions are derived in Section 3.3 to compute the minimum number of registers required to implement a statically scheduled DFG for various memory-sharing models. In Section 3.4, memory minimization is considered simultaneously with retiming, and our conclusions are presented in Section 3.5.

3.2 Preliminaries

The DFG model we consider represents periodic and nonterminating data-flow programs. We consider homogeneous (single-rate) DFGs, where each node is executed once per iteration; however, the techniques used in this chapter can also be applied to multirate DFGs since any well-behaved multirate DFG can be transformed into an equivalent single-rate DFG [55], [56]. Memory requirements for multirate DSP program descriptions have also been considered [57], [58]. In each iteration of the homogeneous DFGs we consider, a node consumes exactly one sample from each arc that is input to the node and produces exactly one sample which is available at the output of the node. Each occurrence of a data path connecting the output of a node to an input of a node is called an arc. Figure 3.2(a) shows one representation of a DFG which contains four arcs, namely arc $U \rightarrow V_1$ with 0 delays, arc $U \rightarrow V_1$ with 4 delays, arc $U \rightarrow V_2$ with 2 delays, and arc $U \rightarrow U$ with 1 delay. Figure 3.2(b) shows another representation of the same DFG. In this chapter, the DFG simply provides a program description. As a result, the two representations in Figures 3.2(a) and (b) can be considered equivalent since they describe the same DSP program.

The DFG is assumed to have no multiplexers and no conditional branches. When computing the number of registers required to implement a DFG, G, it is assumed that

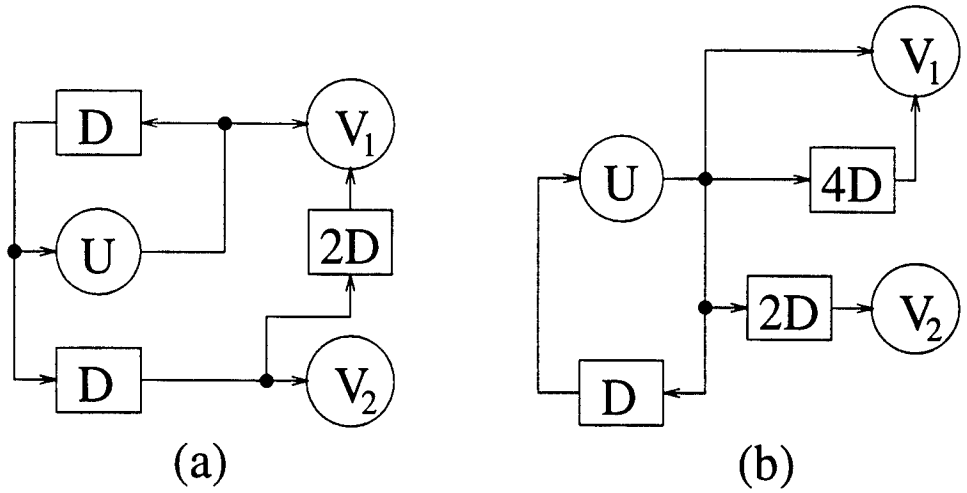


Figure 3.2: (a) A DFG with four arcs. (b) Equivalent representation of the DFG shown in (a).

all arcs in G have both a source node and a sink node in G. Arcs which communicate with the outside world can be included by introducing dummy nodes.

The following subsections describe the pipelined processor model used in this chapter and the systematic folding techniques which form a framework for our derivations.

3.2.1 The Pipelined Processor Model

Consider a processor H with P pipelining stages and computational latency of T units. This pipelined processor is often represented as shown in Figure 3.3(a). The hardware in the dashed box in Figure 3.3(a) is referred to as $H^{(P)}$. A more explicit representation of $H^{(P)}$ is shown in Figure 3.3(b), where the computational latency of each sub-operator $H_1, H_2, \ldots, H_P$ is assumed to be T/P. The dashed box shows that the P delays $D_1, D_2, \ldots, D_P$ are internal to $H^{(P)}$ and cannot be accessed by other data paths.

Consider the implementation of the pipelined processor H shown in Figure 3.3(c). The hardware in the dashed box in Figure 3.3(c) is referred to as $H^{(P')}$. In this case, the $P' = P - 1$ delays $D_1, D_2, \ldots, D_{P-1}$ are internal to $H^{(P')}$, but the delay D_P is external

to $H^{(P')}$ and can be accessed by other data paths. A simplified version of this model is shown in Figure 3.3(d). The structure shown in Figure 3.3(d) may not be acceptable for some applications due to the multiplexer delay, T_{MUX}. The final stage of pipelined processor H has a computational latency of $T_{H_P}+T_{MUX}$, where T_{H_P} is the computational latency of H_P. If $T_{H_P}+T_{MUX}$ is greater than the desired clock period, $T_{DESIRED}$, then the multiplexer must be eliminated and the delay D_P can be dedicated to processor H as in Figure 3.3(b). Throughout this chapter, we assume $T_{H_P}+T_{MUX} \leq T_{DESIRED}$, so that the pipelined processor model $H^{(P')}$ can be used and the delay D_P can be accessed by outputs of other processors, as shown in Figure 3.3(d). We also assume $P \geq 1$ so that P' is nonnegative. When computing the minimum number of registers required to implement a statically scheduled DFG, we do not count the P' registers which are internal to the processor.

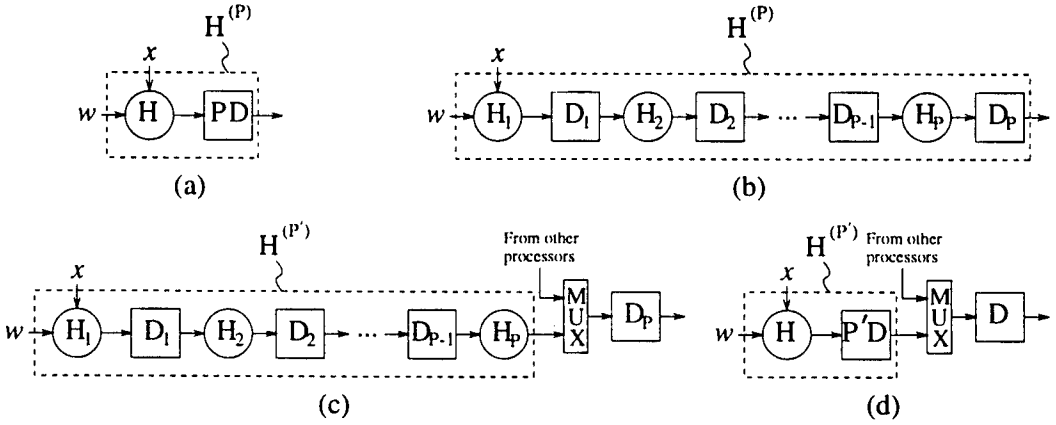


Figure 3.3: (a) Implementation of P-stage pipelined processor H with lumped pipelining delays. (b) Pipelined processor with separated internal pipelining delays. (c) Pipelined processor where the last pipelining delay can be shared with other data paths. (d) A simplified version of (c).

3.2.2 Systematic Folding Techniques

The folding transformation formalized in [28] gives a method of systematically determining control circuit specifications from a statically scheduled DFG. This section presents a brief introduction to these systematic folding techniques.

Consider the algorithm DFG in Figure 3.4(a) which contains the arc $U \rightarrow V$ with i delays. In this system, the result of the l-th iteration of operation U is used for the $(l+i)$-th iteration of operation V. Let N be the folding factor, i.e., N operations are executed using a single hardware operator. Furthermore, let u and v be the folding orders of U and V, respectively. The folding order describes the time partition, or the time unit modulo N, in which an operation is scheduled, i.e., the l-th iteration of U is scheduled to be executed by hardware operator H_U at time unit $(Nl+u)$. Similarly, the $(l+i)$-th iteration of V is scheduled to be executed by hardware operator H_V at time unit $N(l+i)+v$. If H_U has P_U pipelining stages and the pipelined processor model $H^{(P')}$ (see Figure 3.3(d)) is used, then the result of the l-th iteration of U is output from $H_U^{(P')}$ at $(Nl+u+P'_U)$, where $P'_U = P_U - 1$. The folding process maps each arc $U \rightarrow V$ with i delays in the algorithm DFG to an arc in the architecture DFG. We denote by $D_F(U \rightarrow V)$ the number of delays on the arc in the architecture DFG which is the result of folding arc $U \rightarrow V$ in the algorithm DFG. This delay is the difference between the execution time of the $(l+i)$-th iteration of V and the time that the result of the l-th iteration of U is available, i.e.,

$$D_F(U \rightarrow V) = N(l+i)+v-(Nl+u+P'_U) = Ni - P'_U + v - u. \qquad (3.1)$$

Note that the number of folded delays is iteration independent, i.e., $D_F(U \rightarrow V)$ is independent of l. Hardware operator H_U, which is pipelined by P_U stages and has P'_U internal pipelining delays, is connected to hardware operator H_V at switching instance

$(Nl + v)$ with $D_F(U \rightarrow V)$ delays, as shown in Figure 3.4(b). This derivation differs slightly from the derivation in [28] since here we use the pipelined processor model $H^{(P')}$ (see Figure 3.3(d)), where the pipelined processor model $H^{(P)}$ (see Figure 3.3(a)) is used in [28].

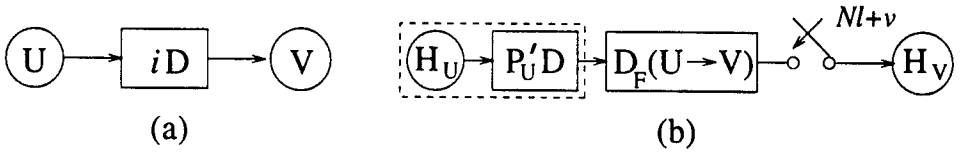


Figure 3.4: (a) An arc $U \rightarrow V$ in the algorithm DFG. (b) The mapping of the folded arc in the architecture DFG.

A folding set is an ordered set of operations which are executed by the same processor. Each folding set contains N entries, some of which may be null operations. The operation in the j-th position within the folding set (where j goes from 0 to $N-1$) is executed by the processor during time partition j. For example, consider the folding set $S_1 = \{A_1, \emptyset, A_2\}$ for $N = 3$. Operation A_1 belongs to folding set S_1 with folding order 0 (also denoted as $(S_1|0)$), and operation A_2 belongs to folding set S_1 with folding order 2 (also denoted as $(S_1|2)$). Due to the null operation in the 1-st position within S_1, the operator that executes operations A_1 and A_2 will not be utilized at time instances $3l + 1$. For a folded system to be realizable, $D_F(U \rightarrow V) \geq 1$ must hold for all arcs. Once valid folding sets have been assigned, pipelining and retiming can be used to satisfy this property (see [28]).

In the folded realization, the data on the system input is assumed to be valid for N clock cycles before changing. For example, if $N = 2$ and the folded realization is assumed to operate with period T, then the input sample $x[0]$ must be valid from 0 to $2T$, $x[1]$ must be valid from $2T$ to $4T$, etc.

We demonstrate the use of systematic folding techniques by folding the biquad filter

in Figure 3.5(a). Assume addition and multiplication require 1 and 2 units of time, respectively (i.e., $T_A = 1$ and $T_M = 2$), and one-stage pipelined adders and two-stage pipelined multipliers are available (i.e., $P_A = 1$ and $P_M = 2$). A retimed version of this filter with valid folding sets assigned using folding factor $N = 4$ is shown in Figure 3.5(b). Folding factor $N = 4$ means that the iteration period of the folded hardware is 4 time units, i.e., each node of the biquad filter is executed exactly once every 4 time units in the folded DFG. The folded circuit is shown in Figure 3.6. To see how the folded DFG in Figure 3.6 is obtained from the algorithm DFG in Figure 3.5(b), consider arc $A_1 \rightarrow M_4$. Using (3.1), we find

$$D_F(A_1 \rightarrow M_4) = 4(2) - 0 + 1 - 3 = 6.$$

This means there is an arc in the folded DFG from the adder to the multiplier with 6 delays. Since this arc ends at node M_4, which has folding order 1 in the algorithm DFG, the folded arc is switched at the input of the multiplier in the folded DFG at $4l + 1$. This folded arc is shaded in Figure 3.6. Using Figure 3.1(a) as another example and assigning folding orders 2 and 8 to the multiply and add operations, respectively, and using $N = 10$ and $P_M = 2$, we get $10(0) - 1 + 8 - 2 = 5$ delays in the folded arc as shown in Figure 3.1(b).

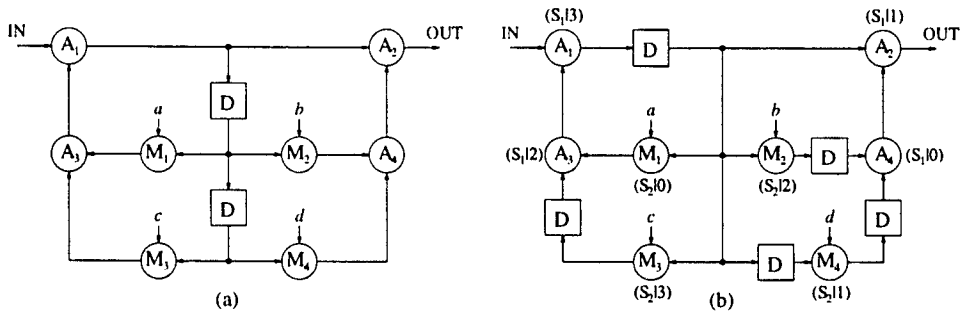


Figure 3.5: (a) The biquad filter. (b) The retimed filter with valid folding sets assigned.

The folded DFG in Figure 3.6 represents the data path specifications obtained from

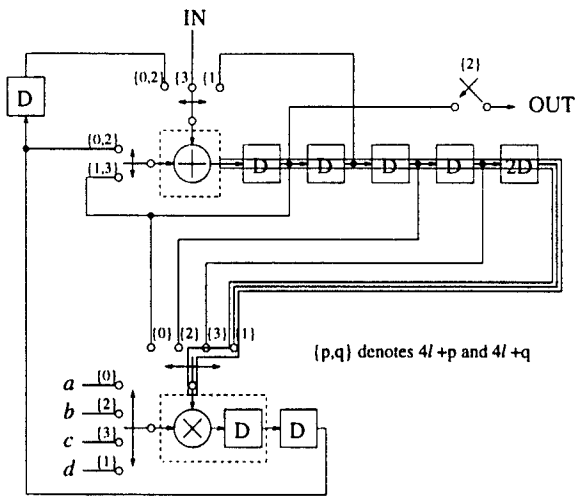


Figure 3.6: The folded biquad filter using the specifications given in Figure 3.5(b). The shaded arc represents arc $A_1 \rightarrow M_4$ in the folded DFG.

the scheduled algorithm DFG by using (3.1); however, this DFG does *not* represent the most efficient implementation of the scheduled DFG in terms of memory usage. Throughout the remaining sections of this chapter, expressions are derived for determining the most efficient implementation of a statically scheduled DFG in terms of the amount of memory required for the implementation. We now introduce some definitions that will be used in these derivations.

Let x_l, $l \geq 0$ be the result of the l-th iteration of operation U. Recall that each node in the DFG is executed exactly once per iteration. Throughout this chapter, we consider only nonnegative iterations of each operation, which results in no loss of generality. Variable x_l is produced exactly once by H_U, but may be consumed multiple times by one or more processors due to the possibility of fanout. We define a unique *production time* and a unique *consumption time* for each variable.

Definition 3.1 *The production time of variable x_l, denoted as p_{x_l}, is the time unit in which x_l is output from $H_U^{(P')}$, which is $Nl + u + P'_U$. The consumption time of x_l,*

denoted as c_{x_l}, is the latest time unit during which x_l is input to any processor.

Recall that u is the folding order of operation U, which is the time partition, or time unit modulo N, in which the operation U is scheduled to be executed by processor H_U. Since we consider only nonnegative iterations of nodes, $p_{x_l} \geq u + P'_U$ always holds. Also, the consumption time must be greater than the production time, i.e., $p_{x_l} < c_{x_l}$ must always hold because $D_F(U \rightarrow V) \geq 1$ is assumed. In the remainder of the chapter, $p_{x_l} \geq u + P'_U$ and $p_{x_l} < c_{x_l}$ are implicitly assumed. We use p_{x_l} and c_{x_l} to define the time interval for which the variable x_l is *live*.

Definition 3.2 *The variable x_l is live for all time units in the interval $(p_{x_l}, c_{x_l}]$.*

3.3 Memory Minimization without Retiming

In this section, we derive expressions for the minimum number of registers required to implement a DFG assuming that the DFG has already been scheduled and no more circuit transformations (e.g., retiming) are to be performed on the DFG. The minimum number of registers required to store the variables that are output from a single node is first computed. The operation-constrained, processor-constrained, and unconstrained memory models are then described, and expressions are derived for the minimum number of registers required to implement arbitrary DFGs under these models.

3.3.1 Minimum Number of Registers for Outputs from a Single Node

Before considering the case where the output variables of a node are broadcast to several arcs (e.g., node U in Figure 3.2), we consider the simple case of a single arc $U \rightarrow V$ as shown in Figure 3.4(a). The minimum number of registers required to implement the $D_F(U \rightarrow V)$ delays in Figure 3.4(b) can be calculated using life-time analysis. If we let

x_l, $l \geq 0$, be the result of the l-th iteration of node U, then the production time of x_l is $p_{x_l} = u + P'_U + Nl$ and its consumption time is $c_{x_l} = p_{x_l} + D_F(U \rightarrow V)$. Consider time unit K. The first variable that is produced by node U is the result of the 0-th iteration of U, and the production time of this variable is defined to be p_{x_0}. A new variable is produced by node U every N time units, so the number of variables which have production times prior to time unit K (i.e., which satisfy $p_{x_l} < K$) is

$$r_{p,U}(K) = \left\lceil \frac{K - p_{x_0}}{N} \right\rceil, \tag{3.2}$$

where $\lceil x \rceil$ is the ceiling of x, which denotes the smallest integer greater than or equal to x. Using a similar argument, the number of these variables with consumption times prior to time unit K (i.e., which satisfy $c_{x_l} < K$) is

$$r_{c,U}(K) = \left\lceil \frac{K - c_{x_0}}{N} \right\rceil. \tag{3.3}$$

Note that these expressions for $r_{p,U}(K)$ and $r_{c,U}(K)$ are valid for all K such that $r_{p,U}(K) \geq 0$ and $r_{c,U}(K) \geq 0$. According to Definition 3.2, a variable is live at time unit K if it is produced prior to K and not consumed prior to K. Therefore, the number of live variables at time unit K is the difference between the number of variables produced prior to time unit K and the number of variables consumed prior to time unit K, i.e., $r_{live,U}(K) = r_{p,U}(K) - r_{c,U}(K)$. Using (3.2) and (3.3), the expression for the number of live variables at time unit K becomes

$$r_{live,U}(K) = \left\lceil \frac{K - p_{x_0}}{N} \right\rceil - \left\lceil \frac{K - c_{x_0}}{N} \right\rceil. \tag{3.4}$$

The minimum number of registers required to implement the $D_F(U \rightarrow V)$ delays in Figure 3.4(b) is the maximum value of $r_{live,U}(K)$ over all K. The value of $r_{live,U}(K)$ is periodic in K with period N because the folded architecture operates periodically with period N. Therefore, we only need to evaluate (3.4) for N consecutive time units.

Evaluating (3.4) at time units $K = qN + n$ for some integer q and $n \in [0, N)$ results in the number of live samples at *time partition* n, given by

$$\begin{aligned} r_{live,U}(n) &= \left\lceil \frac{qN + n - p_{x_0}}{N} \right\rceil - \left\lceil \frac{qN + n - c_{x_0}}{N} \right\rceil \\ &= \left\lceil \frac{n - p_{x_0}}{N} \right\rceil - \left\lceil \frac{n - (p_{x_0} + D_F(U \to V))}{N} \right\rceil, \end{aligned}$$

where $c_{x_0} = p_{x_0} + D_F(U \to V)$ has been used. The minimum number of registers required to implement the $D_F(U \to V)$ delays in Figure 3.4(b) is the maximum value of $r_{live,U}(n)$ over the interval $n \in [0, N)$, i.e.,

$$r_{live,U}^{(max)} = \max_{n \in [0,N)} \{ r_{live,U}(n) \}.$$

The following lemma can be used to find the maximum of $r_{live,U}(n)$ for $n \in [0, N)$.

Lemma 3.1 *Given integers A, B, n, and $N > 0$,*

$$\max_{n \in [0,N)} \left\{ \left\lceil \frac{B + n}{N} \right\rceil - \left\lceil \frac{B - A + n}{N} \right\rceil \right\} = \left\lceil \frac{A}{N} \right\rceil.$$

Proof: Since

$$\left\lceil \frac{B + n}{N} \right\rceil - \left\lceil \frac{B - A + n}{N} \right\rceil \tag{3.5}$$

is periodic in n with period N, we only need to show that the maximum of this expression is $\left\lceil \frac{A}{N} \right\rceil$ for any N consecutive integers. Therefore, it is sufficient to show that

$$\max_{n \in [A-B, A-B+N)} \left\{ \left\lceil \frac{B + n}{N} \right\rceil - \left\lceil \frac{B - A + n}{N} \right\rceil \right\} = \left\lceil \frac{A}{N} \right\rceil.$$

The expression in (3.5) equals $\left\lceil \frac{A}{N} \right\rceil$ for $n = A - B$. It remains to show that

$$\left\lceil \frac{B + n}{N} \right\rceil - \left\lceil \frac{B - A + n}{N} \right\rceil \leq \left\lceil \frac{A}{N} \right\rceil$$

holds for $n = A - B + 1, A - B + 2, \ldots, A - B + N - 1$. This can be written as

$$\left\lceil \frac{A + k}{N} \right\rceil - \left\lceil \frac{k}{N} \right\rceil \leq \left\lceil \frac{A}{N} \right\rceil \tag{3.6}$$

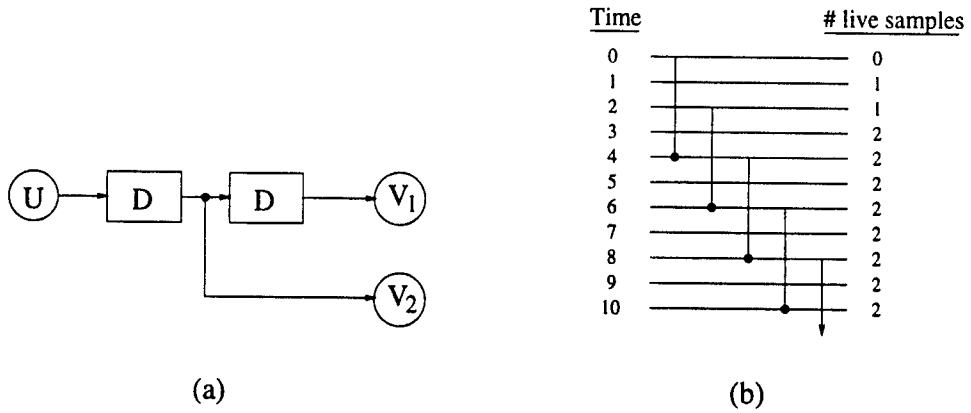


Figure 3.7: (a) A fanout node U. (b) The lifetime chart of samples in the folded architecture.

Table 3.1: Summary of the three memory models described in Section 3.3.2.

memory model	outputs of the nodes executed by the same processor can share registers	outputs of different processors can share registers
operation-constrained (Section 3.3.2)	No	No
processor-constrained (Section 3.3.2)	Yes	No
unconstrained (Section 3.3.2)	Yes	Yes

in G. This results in no loss of generality since arcs that communicate with the outside world can be included by introducing dummy nodes. Let $\mathcal{U}$ be the set of nodes in G with at least one output arc that terminates at a node in G. In this section, the expressions derived in Section 3.3.1 are used to compute the minimum number of registers required to implement G for the operation-constrained, processor-constrained, and unconstrained memory models. Table 3.1 gives an overview of the three memory models discussed in this section.

The Operation-Constrained Memory Model

In the *operation-constrained* memory model, each node $U \in \mathcal{U}$ in G is allocated a unique set of registers in the synthesized hardware. The only variables which are allowed to occupy the registers allocated to U are those variables which result from the execution of node U. As a result, register minimization under the operation-constrained memory model consists of independently computing the minimum number of registers required to implement each node $U \in \mathcal{U}$ and adding these results for all nodes in $\mathcal{U}$. Using (3.10) to compute the number of registers required to implement each node, we get

$$R_O = \sum_{U \in \mathcal{U}} \left\lceil \frac{D_{F,U}^{(max)}}{N} \right\rceil,$$

where $D_{F,U}^{(max)}$ is computed as in (3.8).

Example 3.2 *Consider the scheduled biquad filter in Figure 3.5(b). Recall the assumptions that addition and multiplication require* 1 *and* 2 *units of time, respectively (i.e.,* $T_A = 1$ *and* $T_M = 2$*), and one-stage pipelined adders and two-stage pipelined multipliers are available (i.e.,* $P_A = 1$ *and* $P_M = 2$*). Table 3.2 shows the number of registers required to individually implement each node. For example, the five arcs which are output from node* A_1 *have* 1, 2, 3, 4, *and* 6 *folded arc delays. Since* $\max\{1,2,3,4,6\} = 6$*, node* A_1 *requires* $\lceil 6/4 \rceil = 2$ *registers. By adding the values in Table 3.2, we find* $R_O = 8$*, i.e.,* 8 *registers are required to implement the biquad filter shown in Figure 3.5(b) using the operation-constrained memory model.* □

The operation-constrained memory model is suboptimal with respect to minimization of registers since the registers are often underutilized. For example, consider nodes A_3 and A_4 in Figure 3.5(b). These two nodes belong to folding set S_1 so they are executed by the same processor, which is a one-stage pipelined adder. The outputs of

Table 3.2: The number of registers required to implement the nodes of the biquad filter individually.

Node U	$\left\lceil \frac{D_{F,U}^{(max)}}{N} \right\rceil$
A_1	2
A_3	1
A_4	1
M_1	1
M_2	1
M_3	1
M_4	1

this adder due to A_3 and A_4 must be delayed by 1 time unit since using (3.1) we find that $D_F(A_3 \rightarrow A_1) = 1$ and $D_F(A_4 \rightarrow A_2) = 1$ in Figure 3.5(b). Since the variables resulting from operation A_3 are live during time units $4l + 3$ and the variables resulting from A_4 are live during time units $4l + 1$, these outputs could share the same register; however, under the operation-constrained memory model, each of the nodes A_3 and A_4 requires a separate register. This particular underutilization problem could be eliminated by allowing all variables which are output from the same processor to share registers, which leads to the processor-constrained memory model.

The Processor-Constrained Memory Model

In the *processor-constrained* memory model, each processor in the synthesized hardware is allocated a unique set of registers. The only variables which are allowed to occupy the registers allocated to a processor are those variables which are output from that particular processor. As a result, register minimization under the processor-constrained memory model consists of individually computing the minimum number of registers required to allocate the outputs of each processor and adding these results for all processors.

Recall that the nodes (i.e., operations) which are executed by the same processor belong to the same folding set. The processor-constrained memory model is less restrictive than the operation-constrained memory model since the processor-constrained model allows outputs from the nodes in a folding set to share registers in the synthesized hardware, while the operation-constrained memory model allows no memory sharing among variables produced by different nodes. To determine the number of registers required to implement all nodes in a folding set, we must compute the number of live variables due to the nodes in the folding set for each time partition $n \in [0, N)$.

For each node $U \in \mathcal{U}$, we must first compute $D_{F,U}^{(max)}$ using (3.8). The number of live variables due to node U in time partition n can be found by substituting $p_{x_0} = u + P'_U$ into (3.9) to get

$$r_{live,U}(n) = \left\lceil \frac{n - (u + P'_U)}{N} \right\rceil - \left\lceil \frac{n - (u + P'_U + D_{F,U}^{(max)})}{N} \right\rceil . \tag{3.11}$$

Let $S_1, S_2, \ldots, S_s$ be the folding sets in G. Note that s is the number of folding sets in G, which is equivalent to the number of processors in the folded realization of G. The number of live variables in time partition $n \in [0, N)$ due to all $U \in S_k$ is

$$r_{live,S_k}(n) = \sum_{U \in S_k} r_{live,U}(n),$$

and the number of registers required to implement all nodes $U \in S_k$ is

$$r_{live,S_k}^{(max)} = \max_{n \in [0,N)} \{ r_{live,S_k}(n) \} .$$

The minimum number of registers required to implement G using the processor-constrained memory model is

$$R_P = \sum_{k=1}^{s} r_{live,S_k}^{(max)} = \sum_{k=1}^{s} \left(\max_{n \in [0,N)} \left\{ \sum_{U \in S_k} r_{live,U}(n) \right\} \right) .$$

Table 3.3: The number of live variables at the output of each operator of the folded biquad filter for all possible time partitions.

time	$S_1(+)$	$S_2(\times)$
0	2	2
1	3	1
2	1	2
3	2	1

Example 3.3 *For the biquad filter in Figure 3.5(b), the number of registers required to delay the outputs of the adder is* $r_{live,S_1}^{(max)} = 3$ *and the number of registers required to delay the outputs of the multiplier is* $r_{live,S_2}^{(max)} = 2$. *As a result,* $R_P = 5$, *i.e.,* 5 *registers are required to implement the folded biquad filter using the processor-constrained memory model.*

The processor-constrained memory model may not result in the minimum number of registers because variables which are output from different processors are not allowed to share registers. Table 3.3 shows the number of live variables for the scheduled biquad filter in Figure 3.5(b) for the folding sets S_1 (adder) and S_2 (multiplier) during each time partition. The total number of live variables during any time partition can be found by simply adding the number of live variables due to S_1 and S_2 for that time partition. Notice that the maximum number of live variables in any time partition is 4 even though we computed in Example 3.3 that the folded implementation requires 5 registers using the processor-constrained memory model. This demonstrates that the processor-constrained memory model may not achieve global optimality with respect to register minimization; however, this may still result in an efficient architecture due to local interconnection.

The Unconstrained Memory Model

In the *unconstrained* memory model, each variable can be stored in any register in the synthesized hardware, regardless of the node in the DFG or the processor in the synthesized hardware from which the variable originates. The minimum number of registers required under the unconstrained memory model is computed by taking the maximum of the total number of live variables in G over one period of operation, which can be written as

$$R_U = \max_{n \in [0,N)} \left\{ \sum_{U \in \mathcal{U}} r_{live,U}(n) \right\}, \tag{3.12}$$

where (3.8) and (3.11) are used to compute $r_{live,U}(n)$. The quantity R_U represents the theoretical lower bound on the number of registers required to implement G.

Example 3.4 *Table 3.4 lists the value of* $r_{live,U}(n)$ *for all nodes* $U \in \mathcal{U}$ *and all time partitions* $n \in [0, N)$ *for the biquad filter in Figure 3.5(b). The number of live variables for each time partition can be found by taking the sum of each column, i.e., these values for time partitions* 0, 1, 2, *and* 3 *are* 4, 4, 3, *and* 3, *respectively. The minimum number of registers required using the unconstrained memory model is* $R_U = 4$ *since* $\max\{4, 4, 3, 3\} = 4$. *Recall that, for this example, the operation-constrained memory model required* 8 *registers and the processor-constrained memory model required* 5 *registers.* □

To determine the computational complexity of computing R_U in (3.12), let m be the number of nodes in G. Clearly, the number of nodes $U \in \mathcal{U}$ cannot be greater than m. If we assume the maximum number of inputs to any node is a constant that is independent of m, then the number of arcs in G grows linearly with m, and $D_{F,U}^{(max)}$ in (3.8) can be computed for $U \in \mathcal{U}$ in $O(m)$ time. The maximum number of nodes in G that can be

Table 3.4: The number of live variables due to each node in the biquad filter for all possible time partitions.

	$n=0$	$n=1$	$n=2$	$n=3$
$r_{live,A_1}(n)$	2	2	1	1
$r_{live,A_3}(n)$	0	0	0	1
$r_{live,A_4}(n)$	0	1	0	0
$r_{live,M_1}(n)$	0	0	1	0
$r_{live,M_2}(n)$	1	0	0	0
$r_{live,M_3}(n)$	0	1	1	0
$r_{live,M_4}(n)$	1	0	0	1
$\sum_{U \in \mathcal{U}} r_{live,U}(n)$	4	4	3	3

executed by a single processor is m (the uniprocessor case), so $N \leq m$ holds. Then $r_{live,U}(n)$ in (3.11) can be computed for $U \in \mathcal{U}$ and $n \in [0, N)$ in $O(m^2)$ time. The summation in (3.12) represents $O(m^2)$ additions, and finding the maximum in (3.12) requires $O(m)$ comparisons. Therefore, R_U can be computed for an arbitrary DFG with m nodes in $O(m^2)$ time.

3.3.3 Comparison of Memory Models

Table 3.5 compares the number of registers required for several benchmark filters under the various memory models. The benchmarks used are the fourth-order all-pole lattice filter mentioned in [59] (F1), the fifth-order wave digital elliptic filter introduced in [47] (F2), the fourth-order Jaumann wave digital filter mentioned in [60] (F3), the four-stage pipelined lattice filter [61] (F4), and the biquad filter shown in Figure 3.5(a) (F5). These filters were scheduled using the MARS system [26]. Notice from Table 3.5 that $R_U \leq R_P \leq R_O$ for all of these filters, which appeals to our intuition since the operation-constrained memory model has the most restrictions on memory sharing while the unconstrained memory model has no restrictions on memory sharing.

It is important to note that the three memory models considered in Section 3.3.2 are

Table 3.5: Register count using various memory models. The benchmark filters used are fourth-order lattice filter (F1), fifth-order wave digital elliptic filter (F2), fourth-order Jaumann filter (F3), four-stage pipelined lattice filter (F4), and biquad filter shown in Figure 3.5(a) (F5). N is the iteration period.

Filter	N	R_O	R_P	R_U
F1	10	15	7	6
F2	16	34	12	10
F3	10	16	9	7
F4	2	29	20	18
F5	4	8	5	4

representative of the various models which can be chosen. New memory models can be defined as needed, and expressions can be derived for the minimum number of registers for these models using the same approach as used in Section 3.3.2.

While Table 3.5 gives the number of required registers using the three memory models described in Section 3.3.2, there are side-effects which are not shown in the table. For example, decreasing the number of registers by using the unconstrained model typically increases the number of multiplexers required to allocate data to these registers, and the overall effect of using fewer registers may actually be an increase in area due to the area of the multiplexers. As a result, the number of registers cannot be considered to be the sole cost of the circuit, and several memory models may need to be evaluated to determine the best one for a given application.

3.4 Memory Minimization Using Retiming

The derivations in Section 3.3 are based on the assumption that the DFG has been scheduled and no more circuit transformations are to be performed on the DFG. In this section, we consider optimal retiming of the DFG after scheduling so the resulting implementation uses the minimum number of registers under the unconstrained memory

model.

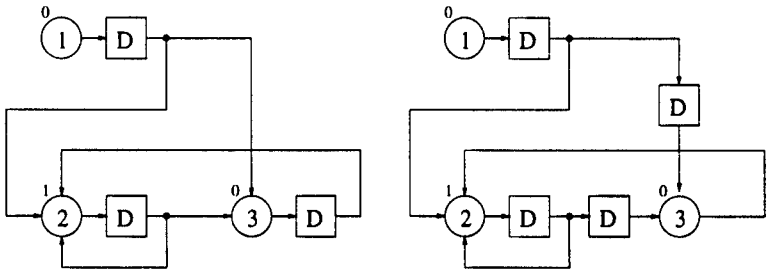
Retiming is often used to reduce the critical path or minimize the number of delays in a circuit [27]. Retiming has also been used for scheduling [11], [12], [26]. This section deals with using retiming to minimize the number of registers in the hardware realization of a statically scheduled DFG. Of course, the retiming must always maintain the validity of the schedule by keeping $D_F(U \rightarrow V) \geq 1$ for all arcs $U \rightarrow V$ so the resulting DFG is realizable.

The problem of minimizing the number of delays in a scheduled DFG is *not* analogous to minimizing the number of registers required by the hardware realization of the DFG. For example, the DFG in Figure 3.8(a) contains 3 delays and its hardware realization requires 5 registers using the unconstrained memory model when we assume an iteration period of $N = 2$ and that all hardware processors are pipelined by $P = 1$ stage. The folding orders are indicated next to the nodes. A retimed version of the DFG is shown in Figure 3.8(b), where the retiming values $r(1) = 0$, $r(2) = 0$, and $r(3) = 1$ are used. This retimed DFG contains 4 delays and its hardware realization requires 4 registers using the unconstrained memory model. From this example, we see that use of retiming to decrease the number of delays in the DFG can actually *increase* the number of registers required to implement the DFG in hardware.

Recall that arc $U \rightarrow V$ in Figure 3.4 is folded using (3.1). Using retiming, the number of delays in arc $U \rightarrow V$ can be changed from i to

$$i_r = i + r(V) - r(U), \tag{3.13}$$

where i_r is the number of delays in arc $U \rightarrow V$ in the retimed algorithm DFG, and $r(X)$ denotes the retiming value of node X [27]. Let $D'_F(U \rightarrow V)$ denote the number of folded arc delays obtained by folding arc $U \rightarrow V$ in the retimed algorithm DFG. To ensure that



(a) (b)

Figure 3.8: (a) A scheduled DFG which has 3 delays and whose hardware requires 5 registers. (b) A retimed version of the DFG which has 4 delays and whose hardware requires 4 registers. For both parts, an iteration period of 2 is assumed and all nodes are mapped to processors with one pipelining stage.

the corresponding arc in the folded hardware DFG has a nonnegative number of delays, we must force the constraint $D'_F(U \rightarrow V) \geq 1$, which is equivalent to

$$Ni_r - P'_U + v - u - 1 \geq 0. \tag{3.14}$$

This constraint ensures that the schedule which was determined prior to retiming is also valid after retiming. Since the retiming values for the nodes are restricted to be integers, (3.13) and (3.14) can be combined as in [28] to obtain

$$r(U) - r(V) \leq \left\lfloor \frac{D_F(U \rightarrow V) - 1}{N} \right\rfloor, \tag{3.15}$$

where $\lfloor x \rfloor$ is the floor of x, which denotes the largest integer less than or equal to x. Once the set of constraints for the DFG is found using (3.15) (there is one such constraint for each arc in the algorithm DFG), a solution must be found using an appropriate technique. We consider an ILP formulation that satisfies the constraints while minimizing the number of registers required to implement the folded hardware DFG.

In addition to the constraints specified by (3.15), the ILP technique must also use constraints to find the maximum values in (3.8) and (3.12). We refer to this formulation as Minimum Physical Storage Location (MPSL) retiming, which is summarized below. The set of equations in Step (II) of MPSL retiming are similar to those used in [21].

MPSL retiming: Minimize R_U subject to

(I) $\forall U \in \mathcal{U}$ and $\forall V \in \mathcal{V}_U$

$$r(U) - r(V) \leq \left\lfloor \frac{D_F(U \rightarrow V) - 1}{N} \right\rfloor$$

(II) $\forall U \in \mathcal{U}$ and $\forall V \in \mathcal{V}_U$

$$D_{F,U}^{'(max)} \geq D_F(U \rightarrow V) + N(r(V) - r(U))$$

(III) $\forall n \in [0, N)$

$$R_U \geq \sum_{U \in \mathcal{U}} \left(\left\lceil \frac{n - (u + P'_U)}{N} \right\rceil - \left\lceil \frac{n - (u + P'_U) - D_{F,U}^{'(max)}}{N} \right\rceil \right)$$

Consider the biquad filter shown in Figure 3.5(a). Assume $T_A = 1$, $T_M = 2$, $P_A = 1$, and $P_M = 2$. The iteration bound, i.e., the lower bound on the achievable iteration period, is 4 units [60], [62], and we consider scheduling the DFG so that the iteration period is equal to the iteration bound. Using the schedule found by the MARS system, the MPSL formulation retimes the DFG such that the minimum number of registers required to implement the biquad filter using the unconstrained memory model is 4. One such retiming of the schedule is shown in Figure 3.5(b) (recall that $R_U = 4$ was computed for Figure 3.5(b) in Example 3.4). Figure 3.9 shows the complete synthesized hardware for the DFG in Figure 3.5(b). Notice that register R_1 is not utilized in time partition 2 and R_4 is not utilized in time partition 3. This underutilization can also be seen in Table 3.4 where the sum of the $n = 2$ and $n = 3$ columns are each equal to 3, so that only 3 of the four registers are utilized during time partitions 2 and 3. In spite of this underutilization, the DFG in Figure 3.5(b) uses the minimum possible number of registers for the given schedule.

The MPSL retiming problem was solved using the ILP solver GAMS [63]. We note

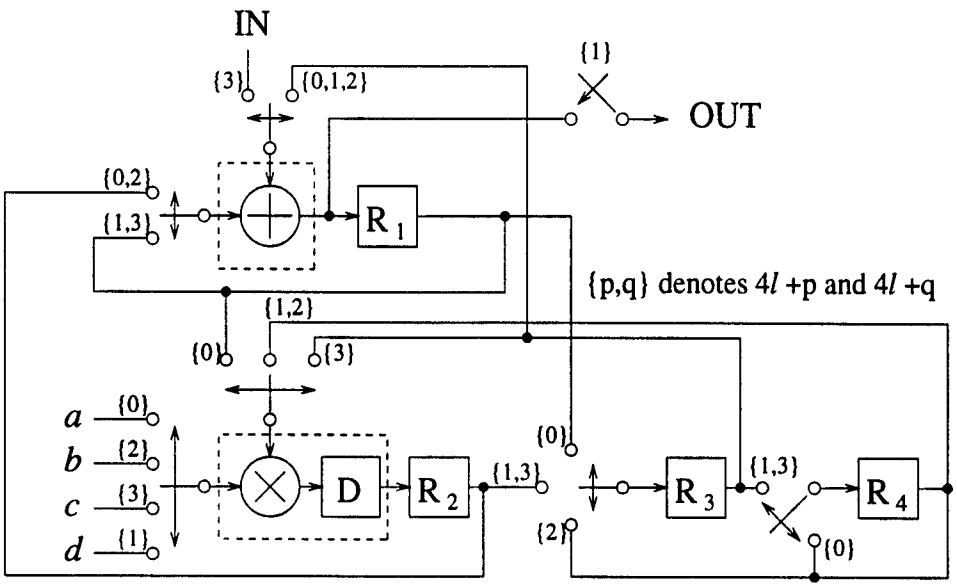


Figure 3.9: The complete synthesized hardware for the scheduled biquad filter in Figure 3.5(b). D and R_i represent word-size registers.

that in some cases, GAMS found an integer solution which it could not prove was optimal. In these cases, we proved that the solution was optimal by showing that there is a time partition for which no better solution exists. When applying MPSL retiming to the schedules obtained by MARS, we found that MPSL retiming did not reduce the number of required registers compared to the retiming performed by MARS, i.e., for the five benchmark filters we considered, the MARS system optimally retimed the filters in terms of the number of registers required under the unconstrained memory model for the schedules generated. Although this result suggests that the retiming performed by MARS is good, it says nothing about the quality of the schedules obtained by MARS with respect to memory requirements.

To determine how the scheduling technique used by the MARS design system performs in terms of minimizing the required number of registers, the MARS schedules were compared to globally optimal results. To determine optimal results in terms of the number of registers, an ILP model is used which schedules a DFG by first minimizing

Table 3.6: Register count for the benchmark filters described in Table 3.5. N is the iteration period. Both scheduling techniques require the minimum number of processors.

Filter	N	MARS schedule using MARS retiming	MARS schedule using MPSL retiming	ILP schedule
F1	10	6	6	5
F2	16	10	10	10
F3	10	7	7	6
F4	2	18	18	18
F5	4	4	4	4

the number of processors and then minimizing the number of registers, as in [37]. The results are shown in Table 3.6, where parameters $T_A = 1$, $P_A = 1$, $T_M = 2$, and $P_M = 2$ are assumed. First, the table shows that MPSL retiming does not change the number of registers required by the MARS schedules. The table also shows that the schedules obtained from the MARS system are optimal or near-optimal in terms of register requirements for the five benchmark filters.

Example 3.5 *Figure 3.10(a) shows a retimed version of the fifth-order wave digital elliptic filter given in [47]. The filter has been retimed using the MPSL retiming according to the schedule in Table 3.7 generated using the MARS system. Figure 3.10(b) shows the synthesized architecture which uses* 10 *registers. The* 10 *registers are denoted as* R_i*, and the internal pipeline delay of the multiplier, which cannot be shared by other data paths, is denoted as* D*. Note that parameters* $T_A = 1$, $P_A = 1$, $T_M = 2$, *and* $P_M = 2$ *are assumed, and the iteration period of the hardware is* 16 *units, which is the iteration bound for the parameters assumed.*

Table 3.7: The schedule from the MARS system for the fifth-order wave digital elliptic filter.

node	1	2	3	4	5	6	7
folding (set|order)	$(S_1\|14)$	$(S_1\|0)$	$(S_1\|11)$	$(S_1\|15)$	$(S_4\|12)$	$(S_1\|10)$	$(S_2\|1)$
node	8	9	10	11	12	13	14
folding (set|order)	$(S_1\|7)$	$(S_2\|11)$	$(S_4\|8)$	$(S_2\|12)$	$(S_2\|15)$	$(S_2\|0)$	$(S_4\|13)$
node	15	16	17	18	19	20	21
folding (set|order)	$(S_1\|6)$	$(S_1\|2)$	$(S_1\|3)$	$(S_2\|7)$	$(S_2\|8)$	$(S_3\|7)$	$(S_4\|4)$
node	22	23	24	25	26	27	28
folding (set|order)	$(S_4\|5)$	$(S_3\|8)$	$(S_3\|2)$	$(S_3\|11)$	$(S_3\|12)$	$(S_4\|9)$	$(S_3\|13)$
node	29	30	31	32	33	34	
folding (set|order)	$(S_3\|0)$	$(S_3\|1)$	$(S_4\|14)$	$(S_1\|12)$	$(S_1\|1)$	$(S_4\|15)$	

3.5 Conclusions

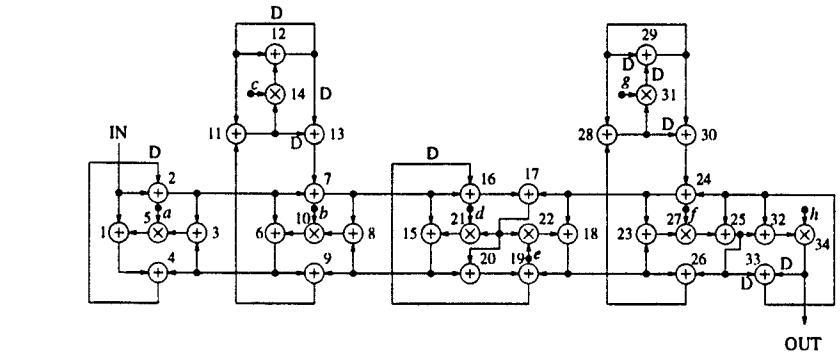
Efficient use of memory in application-specific architectures for DSP is very important in order to meet design specifications. Inefficient use of memory can result in inefficient designs due to effects such as increased area and increased power consumption.

We have derived closed-form expressions for the minimum number of registers required by a statically scheduled DSP program for the operation-constrained, processor-constrained, and unconstrained memory models. We first derived expressions for the minimum number of registers under the operation-constrained and processor-constrained models, and we demonstrated via the biquad filter example why these memory models are not optimal in terms of the number of registers required. We then derived the expression for the minimum number of registers under the unconstrained memory model. This expression, which gives the theoretical lower bound on the number of registers required to implement a statically scheduled DSP program, can be computed in $O(m^2)$ time for

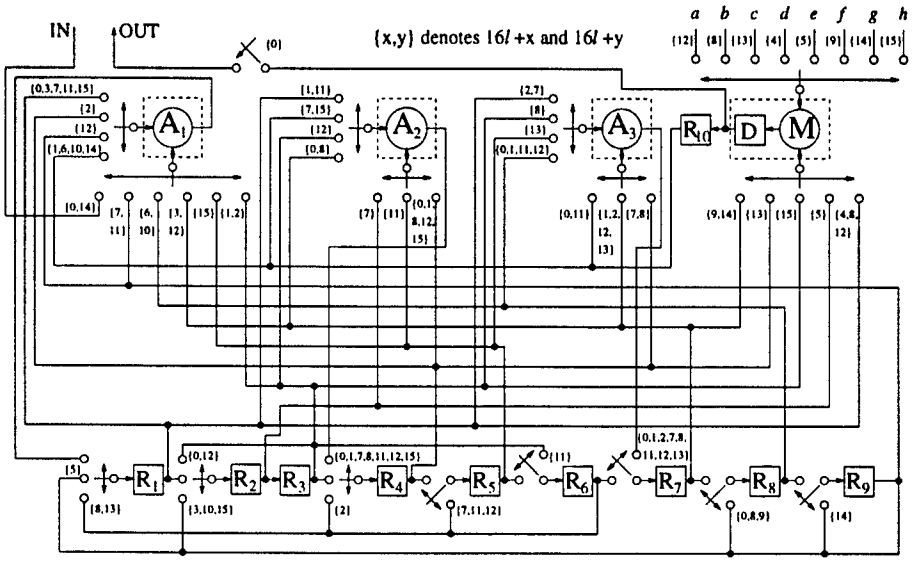
a DFG with m nodes. The techniques we used in our derivations can also be used to determine expressions for lower bounds on memory requirements for other memory models not discussed in the chapter. The results in this chapter are most applicable to dedicated application-specific hardware; however, we believe that these results can also be applied to other technologies, such as FPGA-based designs.

We also considered retiming to minimize memory requirements of a statically scheduled DFG. The MPSL retiming formulation uses integer linear programming techniques to determine the optimal retiming of the DFG in terms of memory required under the unconstrained memory model while maintaining the validity of the schedule. We used MPSL retiming to verify that retiming performed by the MARS system is optimal for the benchmark filters we considered. We then compared memory requirements of schedules obtained by MARS to schedules obtained using integer linear programming which are optimal in terms of required memory under the unconstrained memory model. Our results show that the schedules obtained by MARS are optimal or close to optimal in terms of memory requirements.

The evaluation of the schedules obtained by MARS demonstrates how the techniques presented in this chapter can be used for evaluation of high-level synthesis systems. These techniques can be used for design and evaluation throughout the high-level synthesis process.



(a)



(b)

Figure 3.10: (a) Fifth-order wave digital elliptic filter. The DFG has been retimed using MPSL retiming to minimize the number of registers required given the schedule generated by the MARS system (see Table 3.7). (b) Synthesized hardware using the minimum possible iteration period of 16 and the theoretical lower limit of 10 registers.

Chapter 4

Multirate Folding

4.1 Introduction

The widespread use of digital representation of signals for transmission and storage has created challenges in the area of digital signal processing (DSP). In response to these challenges, new DSP algorithms have emerged for tasks such as compression and filtering of digital signals. Many of these algorithms are *multirate* in nature, meaning that the sample rate is not constant throughout the algorithm description [5]. While the theory of multirate DSP has matured over the past decade, there has been relatively little research on the topic of designing efficient real-time architectures for multirate systems. This has resulted in a lack of CAD tools that can translate multirate algorithms into efficient VLSI architectures.

Considerable work has been done in the area of scheduling multirate DSP algorithms and constructing efficient DSP code for these algorithms [55, 64, 57, 65, 66]. The topic of this chapter is *multirate folding* [36], which is a technique for systematically synthesizing control circuits for single-rate architectures which implement multirate algorithms. Throughout this chapter, the term *single-rate architecture* is used to describe a synchronous architecture where the entire architecture operates with the same clock period.

Examples of data-flow graphs (DFGs) describing multirate DSP algorithms are shown in Figure 4.1. The DFGs in Figure 4.1 are multirate due to decimation by 2 ($\downarrow 2$ block which discards every other sample) and expansion by 2 ($\uparrow 2$ block which inserts a zero between each adjacent pair of samples), which respectively halve and double the sample rate of a signal. A direct mapping of a multirate DSP algorithm to hardware would require data to move at different rates on the chip. This would require routing and synchronization of multiple clock signals on the chip. To avoid these problems, we concentrate on mapping the multirate DSP programs to single-rate VLSI architectures.

The advantages of multirate folding fall into two broad categories. The first advantage is that the multirate folding equations can be used to systematically determine the control circuitry for the architecture from a scheduled DFG. The second advantage, which is slightly more subtle, is that this formal approach can be used to address other related problems in high-level synthesis in a formal manner. Two such problems, memory minimization and retiming [27], are considered in this chapter. Using the multirate folding equations, we derive expressions for the minimum number of registers required to implement the architectures, and we derive constraints for retiming the circuit such that a given schedule is valid.

We first introduced multirate folding in [36] as a technique for synthesizing architectures for tree-structured filter banks. Full and pruned tree-structured filter banks are useful for many DSP applications, such as signal coding and analysis. Recent interest in the discrete wavelet transform (DWT) has significantly increased the number of applications for tree-structured filter banks since the DWT can be computed using a pruned tree-structured filter bank [42, 41, 43, 44]. Computation of wavelet packet bases is another application of pruned tree-structured filter banks [45]. Full binary tree-structured filter banks for signal analysis and synthesis are shown in parts (a) and (b)

of Figure 4.1. Pruned binary tree-structured filter banks which represent analysis and synthesis structures for the discrete wavelet transform (DWT) are shown in parts (c) and (d) of Figure 4.1. Multirate folding can be used to synthesize architectures for each of the four filter banks in Figure 4.1. In Section 4.6, we give a detailed example which shows how the techniques presented in this chapter can be used to design an architecture for the three-level discrete wavelet transform analysis filter bank as shown in Figure 4.1(c).

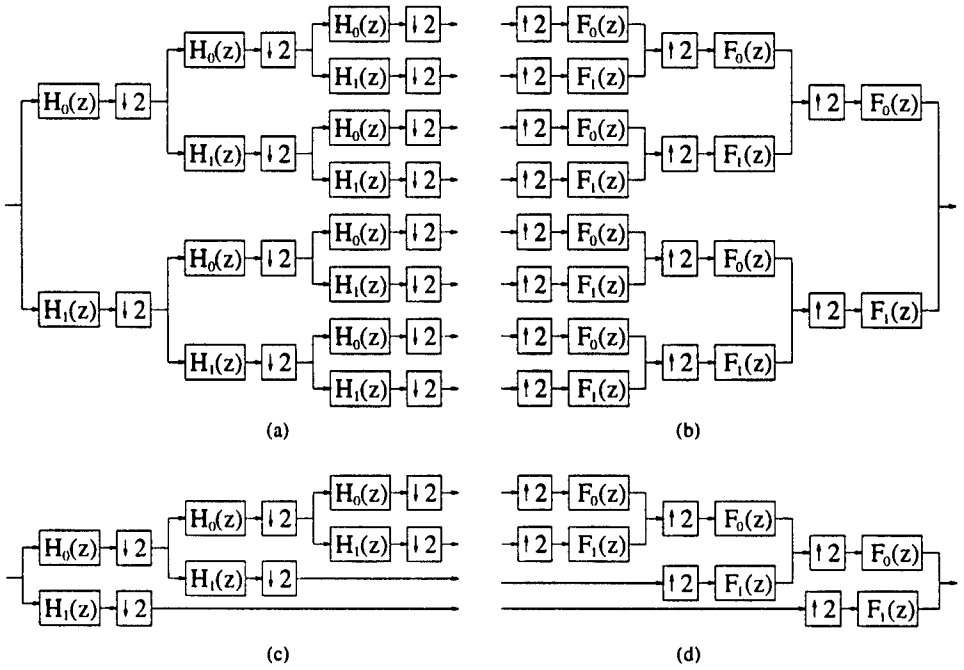


Figure 4.1: Examples of full and pruned binary tree-structured filter banks. (a) Full-tree analysis filter bank. (b) Full-tree synthesis filter bank. (c) Pruned-tree analysis filter bank which can be used to compute the DWT. (d) Pruned-tree synthesis filter bank which can be used to compute the inverse DWT.

The main properties of multirate folding are summarized below:

- Multirate folding is a novel technique for synthesizing control circuits for single-rate architectures which implement multirate DSP algorithms.
- The multirate folding equations allow us to address other problems in high-level

synthesis, such as memory minimization and retiming.

- Multirate folding operates directly on the multirate DFG, avoiding the step of first constructing an equivalent single-rate algorithm description.
- Multirate folding accounts for pipelining, so architectures can be designed for high speed and low power [67] applications.
- Multirate folding is applicable to a wide variety of DSP algorithms. We demonstrate its utility by designing a discrete wavelet transform architecture.

The chapter is organized as follows. Section 4.2 reviews some fundamentals of multirate digital signal processing. In Section 4.3, we derive the folding equations which are used to systematically synthesize the control circuits for the pipelined architectures. Retiming for multirate folding is addressed in Section 4.4. Memory requirements for the folded architectures are addressed in Section 4.5, and the discrete wavelet transform design example is given in Section 4.6. Our conclusions are stated in Section 4.7.

4.2 Some Multirate DSP Fundamentals

This section provides a review of some multirate DSP fundamentals which are used throughout the chapter.

Multirate DSP algorithm descriptions contain decimators and/or expanders. Figure 4.2 shows a decimator and an expander, which obey the input-output relationships $y_D(n) = x(Mn)$ and

$$y_E(n) = \begin{cases} x(\frac{n}{M}) & \text{if } n \text{ is a multiple of } M \\ 0 & \text{otherwise} \end{cases}.$$

Note that we use the term expander rather than interpolator to describe the block in

Figure 4.2(b) since interpolation generally implies expansion followed by filtering. The decimator and expander both have the effect of changing the sample rate.

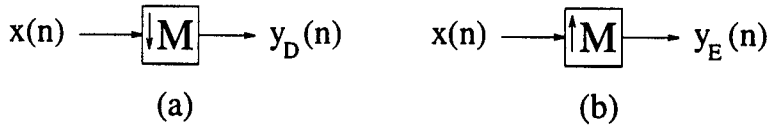


Figure 4.2: (a) Decimation by M. (b) Expansion by M.

The noble identities are useful for theory and implementation of multirate DSP [5]. Special cases of these identities are shown in Figure 4.3. These relationships are used in Section 4.4 to derive conditions for retiming a multirate DFG for folding.

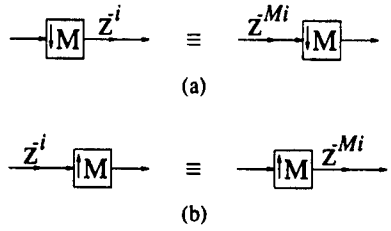


Figure 4.3: Redistribution of delays in a multirate system using the noble identities.

4.3 Derivation of Folding Equations

Folding is a technique for systematically determining control circuits in architectures where multiple algorithm operations (such as addition operations) are time-multiplexed to a single hardware module (such as a pipelined ripple-carry adder) [28]. The folding transformation is similar to loop folding [68] which has been used in high-level synthesis. Figure 4.4 shows an example of how folding can be used to time-multiplex two algorithm operations to a single hardware operator. Folding equations have been derived in the past for folding single-rate algorithms to single-rate architectures, and for folding single-rate algorithms to multirate architectures [28]. In this section, we review folding of single-rate

algorithms to single-rate architectures, and then derive equations for folding multirate algorithms to single-rate architectures.

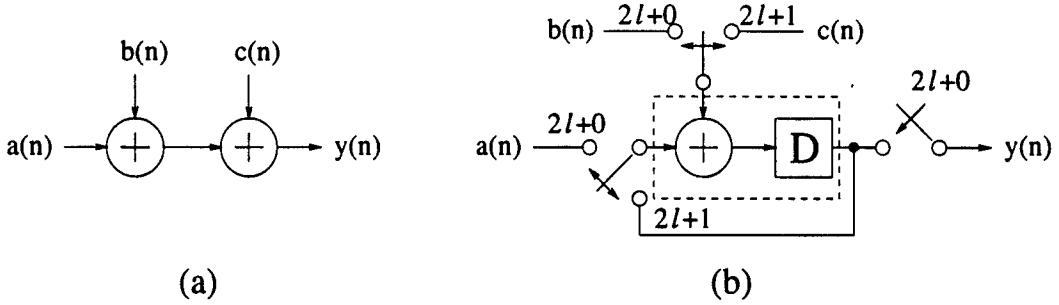


Figure 4.4: (a) A simple single-rate DSP algorithm with two addition operations. (b) A folded architecture where the two addition operations are folded to a single hardware adder with one stage of pipelining.

4.3.1 Single-Rate Folding

Consider an arc (also referred to as an edge) connecting nodes U and V with i delays, as in Figure 4.5(a). Let the l-th iteration of nodes U and V be scheduled to execute at time units $N_U l + u$ and $N_V l + v$, respectively, where u and v are the *folding orders* of nodes U and V which satisfy $u \in [0, N_U)$ and $v \in [0, N_V)$. The hardware operators (also referred to as functional units) which execute nodes U and V are denoted as H_U and H_V, respectively. Note that N_U and N_V number of operations are folded to H_U and H_V, respectively. If H_U is pipelined by P_U stages, then the result of the l-th iteration of node U is available at $N_U l + u + P_U$. Since arc $U \rightarrow V$ has i delays, the result of node U is used by the $(l+i)$-th iteration of V, which is executed at $N_V(l+i) + v$. Therefore, the result must be stored for

$$D_F^S(U \rightarrow V) = N_V(l+i) + v - (N_U l + P_U + u) = (N_V - N_U)l + N_V i - P_U + v - u$$

time units. Since we assume that DSP programs iterate from $l = 0$ to $l = \infty$, practical concerns require $N_U = N_V$ to avoid the cases where $D_F^S(U \rightarrow V)$ approaches $+\infty$ or

$-\infty$ as l gets large. With $N = N_U = N_V$, the folding equation becomes

$$D_F^S(U \to V) = Ni - P_U + v - u, \tag{4.1}$$

which is independent of the iteration number, l. Arc $U \to V$ maps to a path from H_U to H_V in the architecture with $D_F^S(U \to V)$ delays, and data on this path are input to H_V at $Nl + v$, as illustrated in Figure 4.5(b).

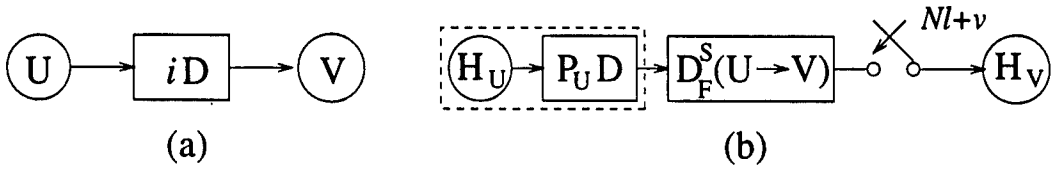


Figure 4.5: (a) An arc $U \to V$ with i delays. (b) The corresponding folded arc.

4.3.2 Multirate Folding

Multirate folding provides a systematic technique for mapping multirate algorithms to single-rate hardware. Folding equations are first derived for arcs which contain decimators and then for arcs which contain expanders.

The Folding Equation for Arcs Containing Decimators

Consider the arc $U \to V$ in Figure 4.6(a), where the output of node U passes through i_1 delays, decimation by M, and i_2 delays before reaching node V. Let the l-th iteration of node U execute at time unit $N_U l + u$ and the l-th iteration of V execute at $N_V l + v$, where the folding orders satisfy $u \in [0, N_U)$ and $v \in [0, N_V)$.

The signals labeled in Figure 4.6(a) are related by

$$\begin{aligned} w_1(l) &= x(l - i_1) \\ w_2(l) &= w_1(Ml) = x(Ml - i_1) \\ y(l) &= w_2(l - i_2) = x(M(l - i_2) - i_1) \end{aligned}$$

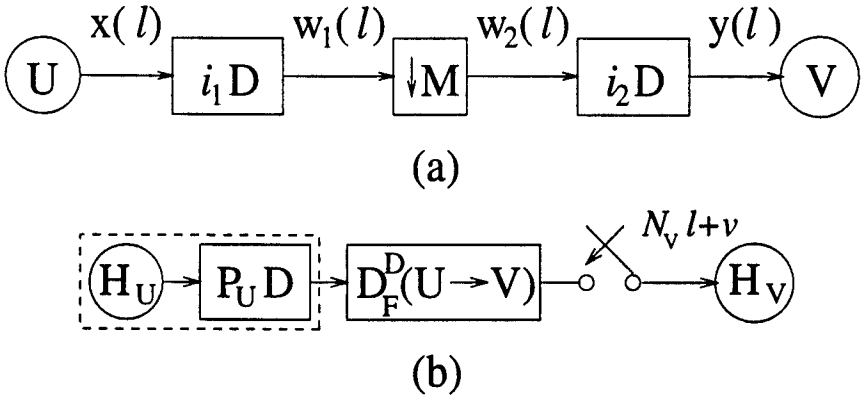


Figure 4.6: (a) An arc $U \rightarrow V$ which contains a decimator. (b) The corresponding folded arc.

which implies that the sample $y(l)$, which is consumed during the l-th iteration of V, is produced during the $(Ml-(Mi_2+i_1))$-th iteration of U. Sample $y(l)$ is consumed by H_V in time unit $N_V l+v$ and is produced by H_U in time unit $N_U(Ml-(Mi_2+i_1))+u$. If H_U is pipelined by P_U stages, then $y(l)$ is available at time unit $N_U(Ml-(Mi_2+i_1))+u+P_U$. Therefore, $y(l)$ must be stored for

$$\begin{aligned} D_F^D(U \rightarrow V) &= N_V l + v - (N_U(Ml - (Mi_2 + i_1)) + u + P_U) \\ &= (N_V - MN_U)l + N_U(Mi_2 + i_1) - P_U + v - u \end{aligned}$$

time units. As in the single-rate case, we would like this expression to be independent of l. This can be achieved by forcing $N_V = MN_U$, which implies that node U executes M times for each execution of node V. This is intuitive since the output of node U is decimated by M before reaching node V. With $N_V = MN_U$, the folding equation becomes

$$D_F^D(U \rightarrow V) = N_U(Mi_2 + i_1) - P_U + v - u, \tag{4.2}$$

which is independent of the iteration number, l.

Since node V is scheduled to execute on hardware operator H_V at time units $N_V l + v$, the data on arc $U \rightarrow V$ are input to H_V at time units $N_V l + v$ as illustrated in

Figure 4.6(b). For the case of $M = 1$, i.e., where the decimator does not affect the data stream, i_1 and i_2 can be combined as $i = i_1 + i_2$, and $N_U = N_V = N$, where N is the iteration period of nodes U and V. Substituting these expressions into (4.2) gives the single-rate folding equation (4.1).

The Folding Equation for Arcs Containing Expanders

Consider the arc $U \rightarrow V$ in Figure 4.7(a), where the output of node U passes through i_1 delays, expansion by L, and i_2 delays before reaching node V. Let the l-th iteration of node U execute at time unit $N_U l + u$ and the l-th iteration of V execute at $N_V l + v$, where the folding orders satisfy $u \in [0, N_U)$ and $v \in [0, N_V)$.

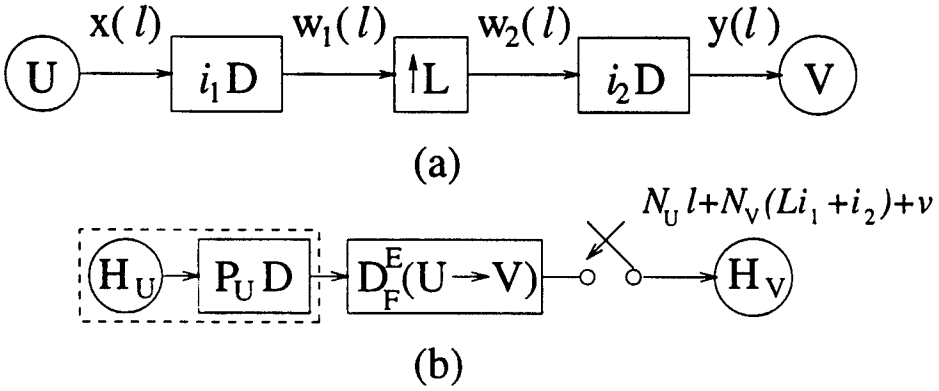


Figure 4.7: (a) An arc $U \rightarrow V$ which contains an expander. (b) The corresponding folded arc.

The signals labeled in Figure 4.7(a) are related by

$$\begin{aligned} w_2(l) &= y(l + i_2) \\ w_1(l) &= w_2(Ll) = y(Ll + i_2) \\ x(l) &= w_1(l + i_1) = y(L(l + i_1) + i_2) \end{aligned}$$

which implies that sample $x(l)$, which is the output of the l-th iteration of U, is used as the input of the $(L(l + i_1) + i_2)$-th iteration of V. Sample $x(l)$ is available at the

output of processor H_U at time unit $N_U l + u + P_U$ and is consumed by H_V at time unit $N_V(L(l + i_1) + i_2) + v$, so $x(l)$ must be stored for

$$\begin{aligned} D_F^E(U \rightarrow V) &= N_V(L(l + i_1) + i_2) + v - (N_U l + u + P_U) \\ &= (N_V L - N_U)l + N_V(Li_1 + i_2) - P_U + v - u. \end{aligned}$$

For this expression to be independent of l, $N_V L = N_U$ must hold. This implies that node V executes L times for every execution of node U, which makes sense since the output of node U is expanded by L before reaching node V. With $N_V L = N_U$, the folding equation becomes

$$D_F^E(U \rightarrow V) = N_V(Li_1 + i_2) - P_U + v - u, \tag{4.3}$$

which is independent of the iteration number, l. The samples on the folded arc are input to H_V at $N_U l + u + P_U + D_F^E(U \rightarrow V) = N_U l + N_V(Li_1 + i_2) + v$, so the folded arc is switched at the input of H_V at $N_U l + N_V(Li_1 + i_2) + v$, as illustrated in Figure 4.7(b).

For the case of $L = 1$, i.e., where the expander does not affect the data stream, i_1 and i_2 can be combined as $i = i_1 + i_2$, and $N_U = N_V = N$, where N is the iteration period of nodes U and V. Substituting these expressions into (4.3) gives the single-rate folding equation (4.1).

4.4 Retiming for Folding

Retiming for folding is the process of retiming a DFG so the number of delays on any folded arc is nonnegative. The constraints which guarantee this for single-rate folding have been derived in [28]. In this section, we review the single-rate constraint and derive the retiming constraints which ensure that the number of folded arc delays is nonnegative for multirate folding.

4.4.1 Single-Rate Case

The constraint which guarantees that the number of folded arc delays is nonnegative for single-rate arcs was derived in [28] to be

$$r(U) - r(V) \leq \left\lfloor \frac{D_F^S(U \rightarrow V)}{N} \right\rfloor. \tag{4.4}$$

This equation is a special case of the constraints which are derived in the next subsection for arcs with decimators or expanders.

4.4.2 Multirate Cases

For (4.2) to be useful, $D_F^D(U \rightarrow V) \geq 0$ must hold given a feasible schedule. The data-flow graph can be retimed to satisfy this condition. Let i_1' and i_2' be the number of delays on arc $U \rightarrow V$ after retiming. Using (4.2), the number of delays on the folded arc after retiming is

$$D_F'^D(U \rightarrow V) = N_U(Mi_2' + i_1') - P_U + v - u.$$

The values of i_1' and i_2' are related to i_1 and i_2 by

$$i_1' = i_1 + Mr(D_{uv}) - r(U)$$

and

$$i_2' = i_2 + r(V) - r(D_{uv}),$$

where $r(u)$ and $r(v)$ are the retiming values of nodes U and V, respectively, i.e., the number of times one delay is removed from each of the output arcs of the node and one delay is added to each of the input arcs of the node. According to multirate DSP fundamentals reviewed in Section 4.2, the retiming value of the decimator, $r(D_{uv})$, is the number of times one delay is removed from its output and M delays are added to

its input. Substituting the expressions for i'_1 and i'_2, we find

$$\begin{aligned} D'^{D}_F(U \rightarrow V) &= N_U[M(i_2 + r(V) - r(D_{uv})) + i_1 \\ &\quad + Mr(D_{uv}) - r(U)] - P_U + v - u \\ &= D^D_F(U \rightarrow V) + N_U(Mr(V) - r(U)), \end{aligned}$$

which is independent of $r(D_{uv})$. We can retime the data-flow graph for folding by forcing $D'^{D}_F(U \rightarrow V) \geq 0$, which gives

$$r(U) - Mr(V) \leq \left\lfloor \frac{D^D_F(U \rightarrow V)}{N_U} \right\rfloor. \tag{4.5}$$

Similarly, we can use retiming to guarantee $D^E_F(U \rightarrow V) \geq 0$, where $D^E_F(U \rightarrow V)$ is computed as in (4.3). If i'_1 and i'_2 are the number of delays on the arc after retiming, then

$$D'^{E}_F(U \rightarrow V) = N_V(Li'_1 + i'_2) - P_U + v - u.$$

The expressions for i'_1 and i'_2 are

$$i'_1 = i_1 + r(E_{uv}) - r(U)$$

and

$$i'_2 = i_2 + r(V) - Lr(E_{uv}),$$

where $r(E_{uv})$ is the retiming value of the expander, which is the number of times we remove L delays from its output and add one delay to its input. Substituting, we get

$$\begin{aligned} D'^{E}_F(U \rightarrow V) &= N_V[L(i_1 + r(E_{uv}) - r(U)) + i_2 + r(V) - Lr(E_{uv})] - P_U + v - u \\ &= D^E_F(U \rightarrow V) + N_V(r(V) - Lr(U)). \end{aligned}$$

as the number of folded arc delays after retiming. Forcing $D'^{E}_F(U \rightarrow V) \geq 0$ gives

$$Lr(U) - r(V) \leq \left\lfloor \frac{D^E_F(U \rightarrow V)}{N_V} \right\rfloor.$$

Caution must be exercised when retiming a multirate DFG due to its periodically time-varying nature. For example, consider the multirate DFG in Figure 4.8(a). If we retime this DFG by assigning the adder a retiming value of -1 and assigning the multiplier a retiming value of 0, we get the DFG in Figure 4.8(b). The problem is that these two circuits have completely different functionality. In the single-rate case, retiming an input node simply results in a delay the output signal, where this example shows that retiming an input node of a multirate DFG can completely change the functionality of the circuit. This issue is taken into consideration in the design example in Section 4.6.

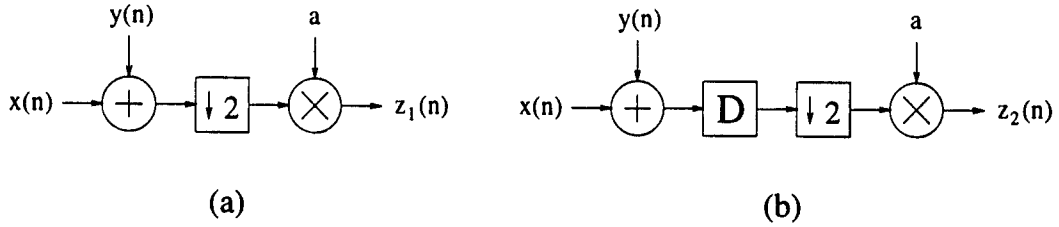


Figure 4.8: (a) A multirate DFG which computes $z_1(n) = a(x(2n)+y(2n))$. (b) Retimed version which computes $z_2(n) = a(x(2n-1)+y(2n-1))$.

4.5 Memory Requirements for Folded DSP Architectures

In this section, we derive expressions for the minimum number of registers required by a folded architecture. The expressions are based on the assumption that a node U in a DFG is one of the following types:

- **Type S:** Each outgoing edge of node U contains no decimators and no expanders.
- **Type D:** Each outgoing edge of node U contains one decimator ($\downarrow M$) and no expanders.
- **Type E:** Each outgoing edge of node U contains no decimators and one expander ($\uparrow L$).

We begin by computing the number of registers required to store the output signal of a Type S node. We then compute the number of registers required to store the output signals of Type E and Type D nodes. Finally, we compute the number of registers required to implement a DSP algorithm which may contain Type S, Type D, and Type E nodes.

4.5.1 Type S Nodes

Consider the simple case of an arc $U \rightarrow V$ as shown in Figure 4.5. The minimum number of registers required to implement the folded edge in Figure 4.5(b) can be calculated using life-time analysis. The idea is to compute the number of samples which exit pipelined processor H_U and enter processor H_V prior to time unit K. By subtracting the number of samples which enter H_V from the number of samples which exit H_U, we find the number of *live* samples at time unit K. The minimum number of registers required to implement the folded edge is the maximum number of live samples over all K.

As in Section 4.3, we assume that the l-th iteration of nodes U and V are scheduled to execute at time units $N_U l + u$ and $N_V l + v$, respectively. We found in Section 4.3 that for this to be feasible $N_U = N_V$ must hold. If we let x_l, $l \geq 0$, be the result of the l-th iteration of node U, then the production time of x_l, which is the time unit that x_l exits pipelined processor H_U in Figure 4.5(b), is $p_{x_l} = N_U l + u + P_U$. The consumption time of x_l, which is the time unit that x_l enters processor H_V, is $c_{x_l} = p_{x_l} + D_F^S(U \rightarrow V)$. The number of samples which have production times prior to time unit K (i.e., which satisfy $p_{x_l} < K$) is

$$r_{p,U}(K) = \left\lceil \frac{K - p_{x_0}}{N_U} \right\rceil, \tag{4.6}$$

where $\lceil x \rceil$ is the ceiling of x, which denotes the smallest integer greater than or equal to x. The number of samples with consumption times prior to time unit K (i.e., which

satisfy $c_{x_l} < K$) is

$$r_{c,U}(K) = \left\lceil \frac{K - c_{x_0}}{N_U} \right\rceil . \tag{4.7}$$

We define x_l to be *live* over the interval $(p_{x_l}, c_{x_l}]$. Using this definition, we find that the number of samples that are live at time unit K is given by $r_{live,U}(K) = r_{p,U}(K) - r_{c,U}(K)$, which is

$$r_{live,U}(K) = \left\lceil \frac{K - p_{x_0}}{N_U} \right\rceil - \left\lceil \frac{K - c_{x_0}}{N_U} \right\rceil . \tag{4.8}$$

The minimum number of registers required to implement the $D_F^S(U \rightarrow V)$ delays in Figure 4.5(b) is the maximum value of $r_{live,U}(K)$ over all K. The value of $r_{live,U}(K)$ is periodic in K with period N_U because the folded architecture operates periodically with period N_U. Therefore, we only need to evaluate (4.8) for N_U consecutive time units. Evaluating (4.8) at time units $K = qN_U + n$ for some integer q and $n \in [0, N_U)$ results in the number of live samples at *time partition* n, given by

$$\begin{aligned} r_{live,U}(n) &= \left\lceil \frac{qN_U + n - p_{x_0}}{N_U} \right\rceil - \left\lceil \frac{qN_U + n - (p_{x_0} + D_F^S(U \rightarrow V))}{N_U} \right\rceil \\ &= \left\lceil \frac{n - p_{x_0}}{N_U} \right\rceil - \left\lceil \frac{n - (p_{x_0} + D_F^S(U \rightarrow V))}{N_U} \right\rceil . \end{aligned}$$

The minimum number of registers required to implement the $D_F^S(U \rightarrow V)$ delays is the maximum value of $r_{live,U}(n)$ over the interval $n \in [0, N_U)$, i.e.,

$$r_{live,U}^{(max)} = \max_{n \in [0, N_U)} \{ r_{live,U}(n) \} .$$

If we let $B = -p_{x_0}$, $A = D_F^S(U \rightarrow V)$, and $N = N_U$, then Lemma 3.1 can be used to show that

$$r_{live,U}^{(max)} = \left\lceil \frac{D_F^S(U \rightarrow V)}{N_U} \right\rceil$$

is the minimum number of registers required to implement the folded edge in Figure 4.5(b).

The more general case, where the output of the node is allowed to be the source of one or more arcs, is now considered. Let $\mathcal{E}_U$ be the set of outgoing edges of node U. We assume for this discussion that node U is a Type S node.

If x_l is an output sample of node U, then the latest time unit in which x_l is scheduled to be used by a processor is

$$c_{x_l} = p_{x_l} + \max_{e \in \mathcal{E}_U} \left\{ D_F^S(U \xrightarrow{e} ?) \right\}. \tag{4.9}$$

If we let

$$D_{F,U}^{S(max)} = \max_{e \in \mathcal{E}_U} \left\{ D_F^S(U \xrightarrow{e} ?) \right\},$$

then (4.9) can be rewritten as

$$c_{x_l} = p_{x_l} + D_{F,U}^{S(max)}.$$

The expressions for $r_{p,U}(K)$ and $r_{c,U}(K)$ for the output signal of node U are the same as in (4.6) and (4.7), and the number of live samples at time unit K is given by (4.8). Substituting $p_{x_0} = u + P_U$, $c_{x_0} = p_{x_0} + D_{F,U}^{S(max)}$, and $K = qN_U + n$ into (4.8) gives the number of live samples at time partition $n \in [0, N_U)$, which is

$$r_{live,U}(n) = \left\lceil \frac{n - p_{x_0}}{N_U} \right\rceil - \left\lceil \frac{n - p_{x_0} - D_{F,U}^{S(max)}}{N_U} \right\rceil. \tag{4.10}$$

Lemma 3.1 can be used to show that the maximum of the expression in (4.10) for $n \in [0, N_U)$ is

$$r_{live,U}^{(max)} = \left\lceil \frac{D_{F,U}^{S(max)}}{N_U} \right\rceil,$$

which is the minimum number of registers required to implement the Type S node.

Example 4.1 *Consider the Type S node in Figure 4.9(a), where the the iteration periods for the nodes are* $N_U = N_{V_1} = N_{V_2} = 2$. *The folding orders for the nodes are* $u = 0$,

$v_1 = 0$, *and* $v_2 = 1$, *and we assume that node* U *is executed by a single-stage pipelined processor, i.e.,* $P_U = 1$. *The folding equations are*

$$\begin{aligned} D_F^S(U \rightarrow V_1) &= 2(2) - 1 + 0 - 0 = 3 \\ D_F^S(U \rightarrow V_2) &= 2(1) - 1 + 1 - 0 = 2, \end{aligned}$$

so $D_{F,U}^{S(max)} = \max\{3, 2\} = 3$. *The minimum number of registers required to implement this Type S node is*

$$\left\lceil \frac{3}{2} \right\rceil = 2.$$

This can also be seen in the lifetime chart in Figure 4.9(b), where the maximum number of live samples for any time step is 2.

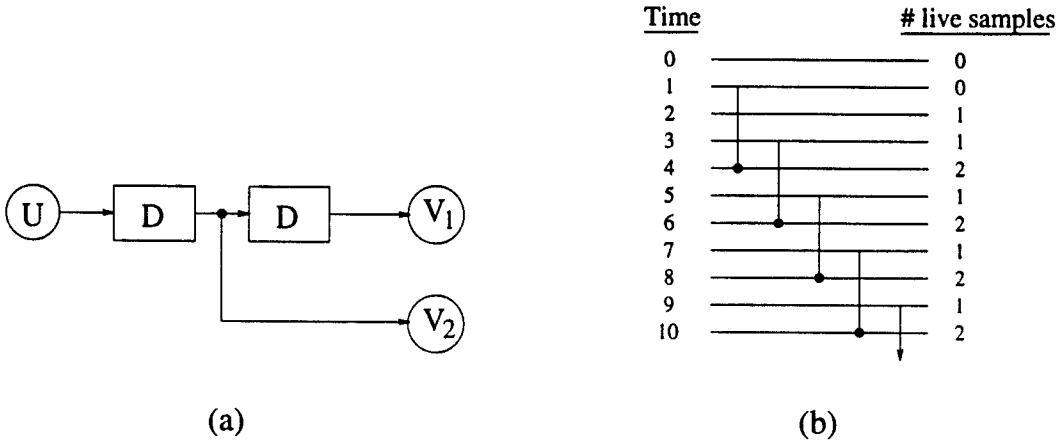


Figure 4.9: (a) A Type S node U. (b) The lifetime chart of samples in the folded architecture.

4.5.2 Type E Nodes

In this section we show how to compute the minimum number of registers required to store the output signal of a Type E node. We begin by computing the minimum number of registers required to implement the folded edge in Figure 4.7(b). Let x_l be the output of the l-th iteration of U, which is available at $p_{x_l} = N_U l + u + P_U$. This sample is

consumed by V at $c_{x_l} = p_{x_l} + D_F^E(U \rightarrow V)$. At time unit K, the number of samples with $p_{x_l} < K$ is

$$r_{p,U}(K) = \left\lceil \frac{K - p_{x_0}}{N_U} \right\rceil .$$

One sample of x_l is produced by node U every N_U time units. Each of these samples is consumed by node V, so one sample of x_l must be consumed by node V every N_U time units in order to avoid a build-up or deficiency of samples of x_l on the folded arc. Since node V consumes one sample of x_l every N_U time units, the number of samples with $c_{x_l} < K$ is

$$r_{c,U}(K) = \left\lceil \frac{K - c_{x_0}}{N_U} \right\rceil ,$$

Keeping Figure 4.7 in mind, it is interesting to note that while U produces a sample of x_l every N_U time units and V consumes a sample of x_l every N_U time units, node V is actually executed in hardware once every $N_V = N_U/L$ time units. As a result, only $(1/L)$-th of the executions of node V in hardware are used to process the output of node U. In a typical multirate system, node V will have L input arcs, each of which occupies $(1/L)$-th of the executions of V in hardware, so all executions of V in hardware are utilized.

The number of live samples of x_l at time unit K is

$$r_{live,U}(K) = \left\lceil \frac{K - p_{x_0}}{N_U} \right\rceil - \left\lceil \frac{K - c_{x_0}}{N_U} \right\rceil . \tag{4.11}$$

Substituting $K = qN_U + n$ and $c_{x_0} = p_{x_0} + D_F^E(U \rightarrow V)$ gives

$$r_{live,U}(n) = \left\lceil \frac{n - p_{x_0}}{N_U} \right\rceil - \left\lceil \frac{n - (p_{x_0} + D_F^E(U \rightarrow V))}{N_U} \right\rceil ,$$

which is the number of live samples of x_j at time partition $n \in [0, N_U)$. Lemma 3.1 can be used to find the minimum number of registers required to implement the folded arc,

which is

$$r_{live,U}^{(max)} = \max_{n\in[0,N_U)} \{r_{live,U}(n)\} = \left\lceil \frac{D_F^E(U \to V)}{N_U} \right\rceil.$$

Computing the memory requirements for a general Type E node, i.e., where the output of node U can be input to several other nodes after expansion by L, is quite simple. Let $\mathcal{E}_U$ be the set of outgoing edges of node U, and let

$$D_{F,U}^{E(max)} = \max_{e\in\mathcal{E}_U} \left\{ D_F^E(U \xrightarrow{e} ?) \right\}.$$

The production time of x_l is $p_{x_l} = N_U l + u + P_U$, and the consumption time is $c_{x_l} = p_{x_l} + D_{F,U}^{E(max)}$. The number of live samples at time unit K is given by (4.11), so we can substitute $K = qN_U + n$ along with expressions for p_{x_0} and c_{x_0} to get

$$r_{live,U}(n) = \left\lceil \frac{n - p_{x_0}}{N_U} \right\rceil - \left\lceil \frac{n - (p_{x_0} + D_{F,U}^{E(max)})}{N_U} \right\rceil,$$

and it follows from Lemma 3.1 that

$$r_{live,U}^{(max)} = \max_{n\in[0,N_U)} \{r_{live,U}(n)\} = \left\lceil \frac{D_{F,U}^{E(max)}}{N_U} \right\rceil.$$

Example 4.2 *Consider the Type E node in Figure 4.10(a) where node U has iteration period $N_U = 6$ and nodes V_1 and V_2 have iteration period $N_{V_1} = N_{V_2} = 2$. The folding orders for the nodes are $u = 2$, $v_1 = 0$, and $v_2 = 1$, and we assume that node U is executed by a single-stage pipelined processor, i.e., $P_U = 1$. The folding equations are*

$$\begin{aligned} D_F^E(U \to V_1) &= 2(3(2) + 0) - 1 + 0 - 2 = 9 \\ D_F^E(U \to V_2) &= 2(3(2) + 1) - 1 + 1 - 2 = 12, \end{aligned}$$

so $D_{F,U}^{E(max)} = \max\{9, 12\} = 12$. The minimum number of registers required to implement this Type E node is

$$\left\lceil \frac{12}{6} \right\rceil = 2.$$

This can also be seen in the lifetime chart in Figure 4.10(b), where the maximum number of live samples for any time step is 2.

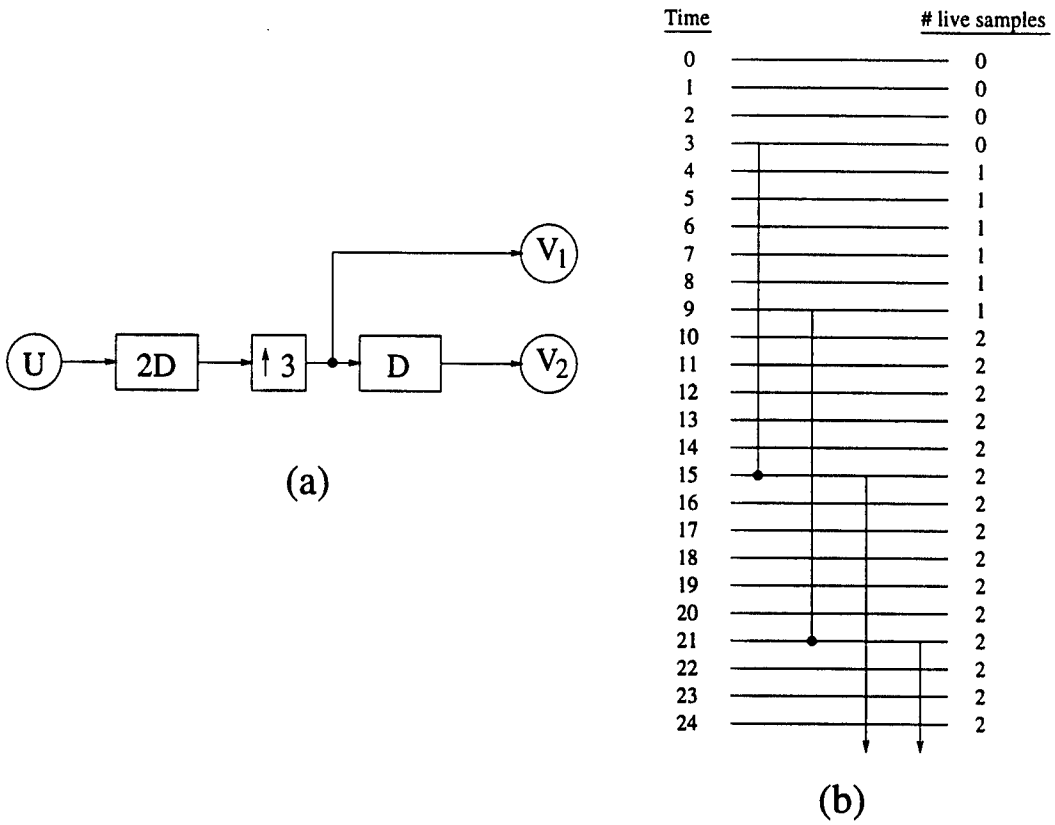


Figure 4.10: (a) A Type E node U. (b) The lifetime chart of samples in the folded architecture.

4.5.3 Type D Nodes

In this section we show how to compute the minimum number of registers required to store the output signal of a Type D node. We begin by computing the minimum number of registers required to implement the folded edge in Figure 4.6(b). Let x_l, $l \geq 0$, be the result of the l-th iteration of U. The first step is to partition x_l into M subsequences $x_j^m = x_{Mj+m}$ for $j \geq 0$ and $m \in [0, M)$. We must now determine which of these M subsequences of x_l is consumed by node V. To determine this, recall that $y(k) = x(M(k - i_2) - i_1)$ in Figure 4.6(a). This can be rewritten as $y(k) = x(Mk_2 + k_1)$ where

$$k_2 = k - i_2 - \left\lceil \frac{i_1}{M} \right\rceil$$

and

$$k_1 = M\left\lceil \frac{i_1}{M} \right\rceil - i_1.$$

Notice here that $0 \leq k_1 \leq M-1$ always holds. Based on this analysis, we can see that node V in Figure 4.6(a) consumes the subsequence $x_j^m = x_{Mj+m}$ for $j \geq 0$ and $m = M\left\lceil \frac{i_1}{M} \right\rceil - i_1$.

Sample x_j^m is output from pipelined processor H_U at time unit $p_{x_j^m} = N_U(Mj+m) + u + P_U$. This sample is input to processor H_V at time unit $c_{x_j^m} = p_{x_j^m} + D_F^D(U \to V)$. One can see from these expressions that one sample of x_j^m is produced and consumed every $N_V = MN_U$ time units. At time unit K, the number of samples of x_j^m with $p_{x_j^m} < K$ is

$$r_{p,U_m}(K) = \left\lceil \frac{K - p_{x_0^m}}{N_V} \right\rceil,$$

and the number with consumption times satisfying $c_{x_j^m} < K$ is

$$r_{c,U_m}(K) = \left\lceil \frac{K - c_{x_0^m}}{N_V} \right\rceil.$$

The number of live samples of x_j^m at time unit K is

$$r_{live,U_m}(K) = \left\lceil \frac{K - p_{x_0^m}}{N_V} \right\rceil - \left\lceil \frac{K - c_{x_0^m}}{N_V} \right\rceil. \tag{4.12}$$

Substituting $K = qN_V + n$ for integer q and $n \in [0, N_V)$ and $c_{x_0^m} = p_{x_0^m} + D_F^D(U \to V)$ gives

$$r_{live,U_m}(n) = \left\lceil \frac{n - p_{x_0}}{N_V} \right\rceil - \left\lceil \frac{n - (p_{x_0} + D_F^D(U \to V))}{N_V} \right\rceil.$$

Using Lemma 3.1, we find that the number of registers required to implement the folded edge in Figure 4.6(b) is

$$r_{live,U}^{(max)} = \max_{n \in [0,N_V)} \{r_{live,U}(n)\} = \left\lceil \frac{D_F^D(U \to V)}{N_V} \right\rceil.$$

We now consider the memory requirements for a general Type D node, where the output of node U may be the input to several nodes. Let $\mathcal{E}_{U_m}$ denote the set of outgoing

edges of node U which are incident into nodes which consume the subsequence x_j^m. In other words, each edge $e \in \mathcal{E}_{U_m}$ satisfies $M\left\lceil \frac{i_1}{M} \right\rceil - i_1 = m$, where i_1 is the number of delays on e between U and the decimator on e.

The number of live samples at time unit K for the edges in $\mathcal{E}_{U_m}$ is given by (4.12). The production time of x_j^m is still $p_{x_j^m} = N_U(Mj+m)+u+P_U$. The consumption time is now $c_{x_j^m} = p_{x_j^m} + D_{F,U_m}^{D(max)}$, where

$$D_{F,U_m}^{D(max)} = \max_{e \in \mathcal{E}_{U_m}} \left\{ D_F^D(U \xrightarrow{e} ?) \right\}. \tag{4.13}$$

Using these expressions along with $K = qN_V + n$ in (4.12) gives

$$r_{live,U_m}(n) = \left\lceil \frac{n - (N_U m + u + P_U)}{N_V} \right\rceil - \left\lceil \frac{n - (N_U m + u + P_U + D_{F,U_m}^{D(max)})}{N_V} \right\rceil \tag{4.14}$$

as the number of live samples of subsequence x_j^m at time partition $n \in [0, N_V)$.

The minimum number of registers required to implement the edges in $\mathcal{E}_{U_m}$ is

$$r_{live,U_m}^{(max)} = \max_{n \in [0,N_V)} \left\{ r_{live,U_m}(n) \right\}.$$

Lemma 3.1 can be used to show that

$$r_{live,U_m}^{(max)} = \left\lceil \frac{D_{F,U_m}^{D(max)}}{N_V} \right\rceil. \tag{4.15}$$

The amount of memory required to store x_{Ml+m} can be determined using (4.15) for each $m \in [0, M)$. Therefore, one might mistakenly assume that the number of registers required to store all output samples of U is the *sum* of the *minimum* number of registers required to store each of the M subsequences x_j^m, i.e., an *incorrect* expression for the minimum number of registers required to store the output samples of node U is

$$\sum_{m=0}^{M-1} \left\lceil \frac{D_{F,U_m}^{D(max)}}{N_V} \right\rceil. \tag{4.16}$$

The *correct* technique is to find the *maximum* value over $n \in [0, N_V)$ of the *sum* of the number of live samples for the M subsequences x_j^m. Therefore, to examine the *total* number of live samples at time partition $n \in [0, N_V)$, we use

$$r_{live,U}(n) = \sum_{m=0}^{M-1} r_{live,U_m}(n), \tag{4.17}$$

and take the maximum of this expression. Combining (4.17) with (4.14) results in

$$r_{live,U}(n) = \sum_{m=0}^{M-1} \left(\left\lceil \frac{n - (N_U m + u + P_U)}{N_V} \right\rceil - \left\lceil \frac{n - (N_U m + u + P_U + D_{F,U_m}^{D(max)})}{N_V} \right\rceil \right). \tag{4.18}$$

The minimum number registers required to store the output samples of node U is the maximum of $r_{live,U}(n)$ over the interval $[0, N_V)$, given by

$$r_{live,U}^{(max)} = \max_{n \in [0, N_V)} \{ r_{live,U}(n) \}. \tag{4.19}$$

We now summarize the technique for determining the minimum number of registers required to implement the output of a Type D node.

1. Partition the outgoing edges of node U into M sets $\mathcal{E}_{U_m}$, where an edge $e \in \mathcal{E}_{U_m}$ has i_1 delays between U and the decimator on e, and $M \left\lceil \frac{i_1}{M} \right\rceil - i_1 = m$ holds.

2. Compute the quantity in (4.13) for $m \in [0, M)$.

3. Compute the minimum number of registers using (4.18) and (4.19).

Example 4.3 *In this example we compute the memory requirements for the Type D node in Figure 4.11. The iteration periods of the nodes are* $N_U = 2$ *and* $N_{V_i} = 6$ *for* $i = 0, 1, 2, 3$. *The folding orders are* $u = 1$, $v_0 = 1$, $v_1 = 2$, $v_2 = 4$, *and* $v_3 = 5$. *Node* U *is assigned to a processor which is pipelined by one stage, i.e.,* $P_U = 1$. *Let* e_i *be the label of the edge from node* U *to node* V_i, *i.e., the four edges of the DFG are* $U \xrightarrow{e_i} V_i$

for $i = 0, 1, 2, 3$. *Recall that* $\mathcal{E}_{U_m}$ *is the set of edges which connect node* U *to nodes which consume samples* $x(Ml+m)$, $m \in [0, M)$, *where* $x(n)$ *is the output of node* U, *so* $\mathcal{E}_{U_0} = \{e_2, e_3\}$, $\mathcal{E}_{U_1} = \{e_0\}$, *and* $\mathcal{E}_{U_2} = \{e_1\}$. *The folding equations are*

$$\begin{aligned} D_F^D(U \xrightarrow{e_0} V_0) &= 2(3(1)+2) - 1 + 1 - 1 = 9 \\ D_F^D(U \xrightarrow{e_1} V_1) &= 2(3(0)+1) - 1 + 2 - 1 = 2 \\ D_F^D(U \xrightarrow{e_2} V_2) &= 2(3(0)+3) - 1 + 4 - 1 = 8 \\ D_F^D(U \xrightarrow{e_3} V_3) &= 2(3(2)+0) - 1 + 5 - 1 = 15, \end{aligned}$$

and the values of $D_{F,U_m}^{(max)}$ *are as shown in Table 4.1.*

Table 4.1: Values of $D_{F,U_m}^{(max)}$ for Example 4.3.

m	0	1	2
$D_{F,U_m}^{(max)}$	15	9	2

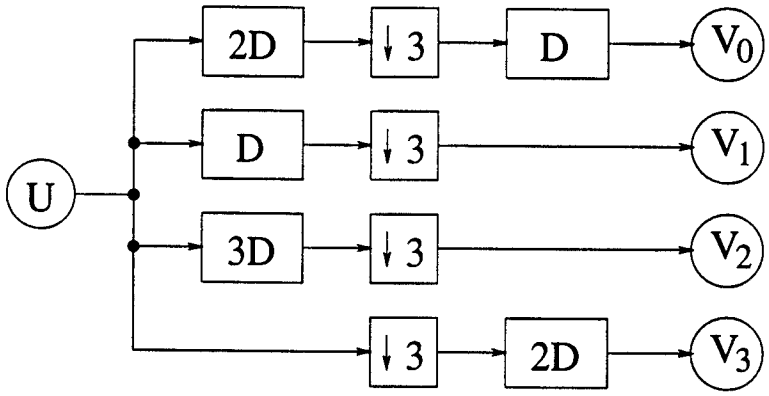


Figure 4.11: A Type D node U with several fanout arcs.

The correct way to compute the minimum number of registers is to use (4.19), which for this example is

$$r_{live,U}^{(max)} = \max_{n \in [0,6)} \left\{ \sum_{m=0}^{2} \left(\left\lceil \frac{n - (2m+1+1)}{6} \right\rceil - \left\lceil \frac{n - (2m+1+1+D_{F,U_m}^{D(max)})}{6} \right\rceil \right) \right\}$$

$$= \max_{n\in[0,6)} \left\{ \left\lceil \frac{n-2}{6} \right\rceil - \left\lceil \frac{n-17}{6} \right\rceil + \left\lceil \frac{n-4}{6} \right\rceil - \left\lceil \frac{n-13}{6} \right\rceil + \left\lceil \frac{n-6}{6} \right\rceil - \left\lceil \frac{n-8}{6} \right\rceil \right\}$$
$$= \max\{4,5,4,4,4,5\} = 5.$$

To see that (4.16) does not compute the minimum number of registers, note that (4.16) gives

$$\sum_{m=0}^{2} \left\lceil \frac{D_{F,U_m}^{D(max)}}{6} \right\rceil = 3+2+1 = 6,$$

which is one larger than the minimum number of required registers.

The lifetime chart [51] which verifies that 5 registers are required is shown in Figure 4.12.

4.5.4 Memory requirements for a general DFG

Consider a DFG, where a node in the DFG can be a Type S, Type D, or Type E node. Let $\mathcal{U}$ denote the set of nodes in the DFG which are Type S, Type D, or Type E nodes. Based on the derivations of this section, we can write the expression for the number of live samples in the folded architecture for time unit n as

$$r_{live}(n) = \sum_{U\in\mathcal{U}} r_{live,U}(n), \tag{4.20}$$

where the expressions for $r_{live,U}(n)$ are summarized in Table 4.2. The minimum number of registers required to implement this architecture is the maximum value of $r_{live}(n)$ over the interval $[0, N_{lcm})$, where N_{lcm} is the least common multiple of the denominators of all of the ceiling functions in (4.20). These concepts are now demonstrated in the following example. This example is intended to demonstrate the memory minimization techniques for multirate folding that are introduced in this section. Examples which demonstrate how to use multirate folding to synthesize useful architectures, such as those for M-ary tree structured filter banks, are given in Section 4.6 and in [36].

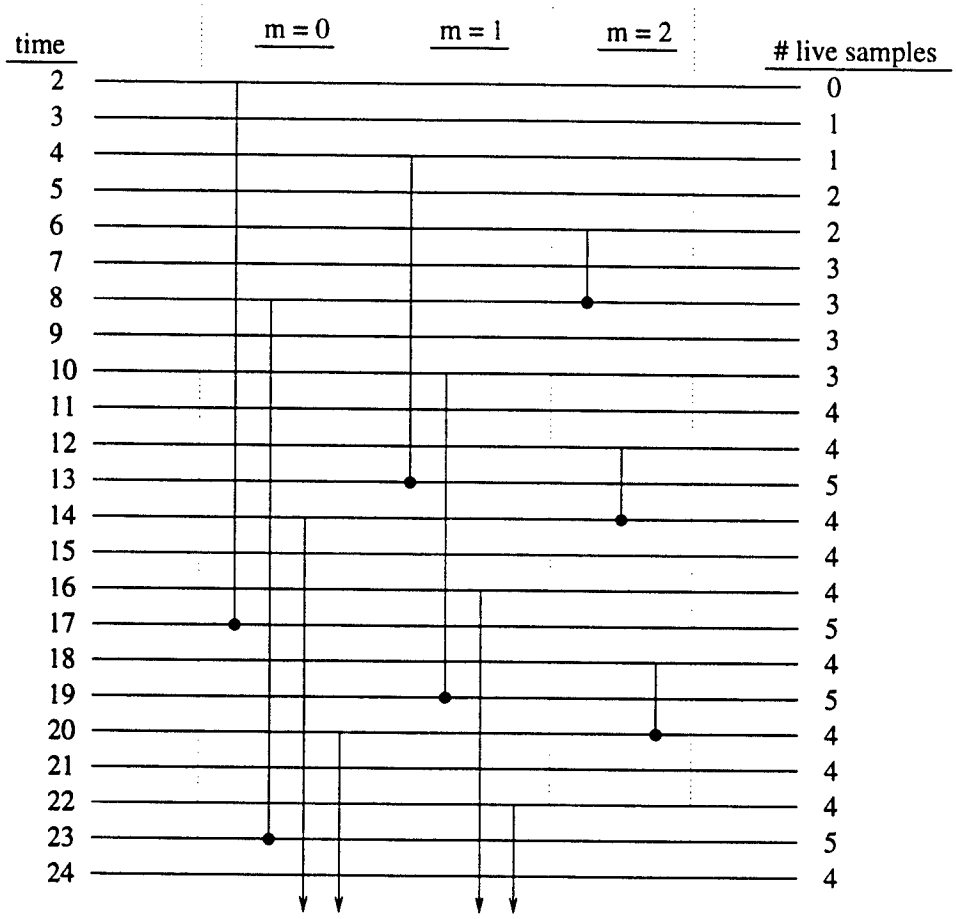


Figure 4.12: The lifetime chart for Example 4.3. The folded implementation requires 5 registers since this is the maximum number of live samples at any time step.

Example 4.4 *Consider the multirate DFG in Figure 4.13. In this figure, A is a Type D node, B and C are Type S nodes, D and E are Type E nodes, and F is a sink node. The iteration periods for the nodes are* $N_A = N_F = 1$ *and* $N_B = N_C = N_D = N_E = 2$. *The folding orders are* $a = 0$, $b = 1$, $c = 0$, $d = 0$, $e = 1$, *and* $f = 0$. *Each node is executed in hardware by a processor which is pipelined by one stage, so* $P_A = P_B = P_C = P_D = P_E = P_F = 1$. *In the architecture, nodes B and C are time multiplexed to the same processor, and nodes D and E are time multiplexed to the same processor. Based*

Table 4.2: Summary of the expressions for $r_{live,U}(n)$ for the various types of nodes. Note that u is the folding order of node U, and P_U is the number of pipelining stages in hardware unit H_U which executes node U.

Node Type	Expression for $r_{live,U}(n)$
S	$\left\lceil \frac{n-(u+P_U)}{N} \right\rceil - \left\lceil \frac{n-(u+P_U+D_{F,U}^{S(max)})}{N} \right\rceil$
D	$\sum_{m=0}^{M-1} \left(\left\lceil \frac{n-(N_U m+u+P_U)}{N_V} \right\rceil - \left\lceil \frac{n-(N_U m+u+P_U+D_{F,U_m}^{D(max)})}{N_V} \right\rceil \right)$
E	$\left\lceil \frac{n-(u+P_U)}{N_U} \right\rceil - \left\lceil \frac{n-(u+P_U+D_{F,U}^{E(max)})}{N_U} \right\rceil$

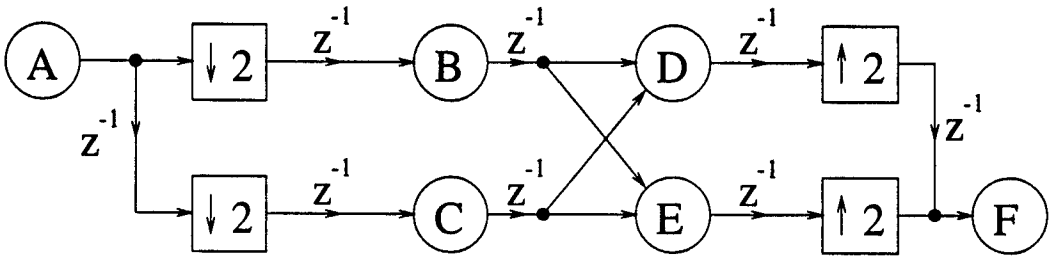


Figure 4.13: Multirate DFG for Example 4.4.

on these parameters, the folding equations are

$$D_F^D(A \to B) = 1(2(1)+0) - 1 + 1 - 0 = 2$$

$$D_F^D(A \to C) = 1(2(1)+1) - 1 + 0 - 0 = 2$$

$$D_F^S(B \to D) = 2(1) - 1 + 0 - 1 = 0$$

$$D_F^S(B \to E) = 2(1) - 1 + 1 - 1 = 1$$

$$D_F^S(C \to D) = 2(1) - 1 + 0 - 0 = 1$$

$$D_F^S(C \to E) = 2(1) - 1 + 1 - 0 = 2$$

$$D_F^E(D \to F) = 1(2(1)+1) - 1 + 0 - 0 = 2$$

$$D_F^E(D \to F) \;=\; 1(2(1)+0)-1+0-1=0.$$

The maximum fanout values are $D_{F,A_0}^{D(max)}=2$, $D_{F,A_1}^{D(max)}=2$, $D_{F,B}^{S(max)}=1$, $D_{F,C}^{S(max)}=2$, $D_{F,D}^{E(max)}=2$, *and* $D_{F,E}^{S(max)}=0$. *Recall that node A has two maximum fanout values (for* $m=0$ *and* $m=1$*) because it is a Type D node with decimation by* $M=2$ *on each of its output arcs.*

The number of live samples at time partition n is given by

$$r_{live}(n) = \sum_{U\in\{A,B,C,D,E\}} r_{live,U}(n)$$

which is

$$\begin{aligned} r_{live}(n) \;=\; & \left\lceil\frac{n-1}{2}\right\rceil - \left\lceil\frac{n-3}{2}\right\rceil + \left\lceil\frac{n-2}{2}\right\rceil - \left\lceil\frac{n-4}{2}\right\rceil + \left\lceil\frac{n-2}{2}\right\rceil - \left\lceil\frac{n-3}{2}\right\rceil \\ & + \left\lceil\frac{n-1}{2}\right\rceil - \left\lceil\frac{n-3}{2}\right\rceil + \left\lceil\frac{n-1}{2}\right\rceil - \left\lceil\frac{n-3}{2}\right\rceil + \left\lceil\frac{n-2}{2}\right\rceil - \left\lceil\frac{n-2}{2}\right\rceil, \end{aligned}$$

where the first two terms are for A_0, *the next two for* A_1 *followed by two terms each for nodes B, C, D, and E. The minimum number of registers required for the architecture is*

$$r_{live}^{(max)} = \max_{n\in\{0,1\}}\{r_{live}(n)\} = \max\{4,5\} = 5.$$

One implementation which uses 5 registers is shown in Figure 4.14, where processor P_1 *executes node A, processor* P_2 *executes nodes B and C, processor* P_3 *executes nodes D and E, and processor* P_4 *executes node F.*

4.6 Design Example

In this section we give an example which illustrates how the folding equations, retiming for folding constraints, and memory minimization can be used to synthesize a single-rate architecture for a multirate DSP algorithm. The algorithm we consider is the three-level

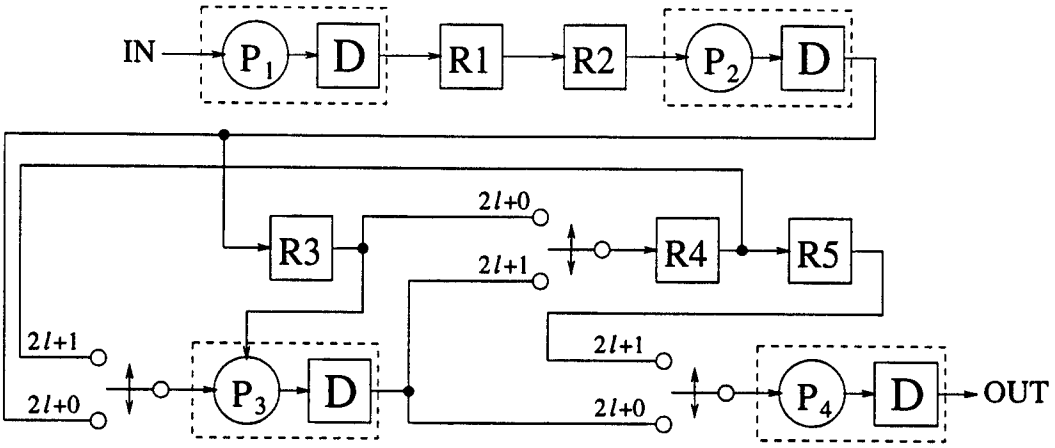


Figure 4.14: Folded architecture for Example 4.4. D denotes an internal pipelining delay, while R_i denote external registers. This implementation uses five registers, which is the minimum value computed in the example.

orthogonal discrete wavelet transform analysis filter bank which uses third-order wavelet filters, as shown in Figure 4.15 [5]. The schedule for the architecture is given in Table 4.3.

The steps we take in deriving the folded architecture are as follows:

1. Write the folding equations for the DFG.
2. Write the retiming-for-folding constraints and find a solution.
3. Write the folding equations for the retimed DFG.
4. Determine the memory requirements for the folded architecture.
5. Allocate data to the minimum number of registers.
6. Draw the folded architecture.

Each of these steps is described in detail in the following subsections.

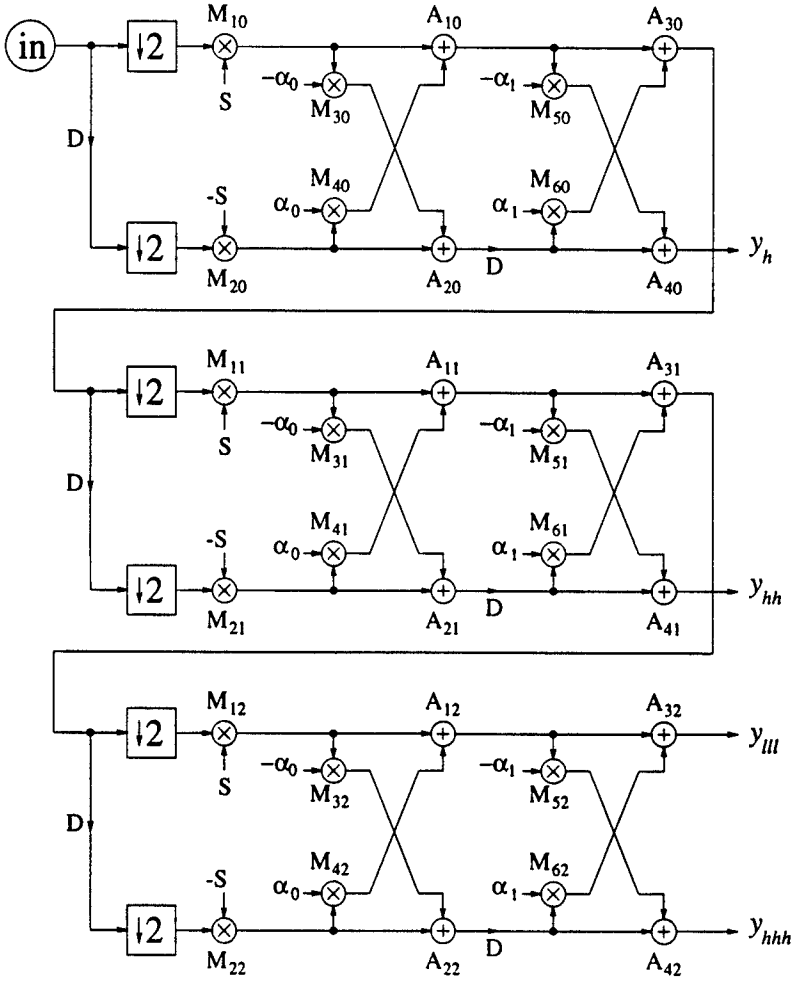


Figure 4.15: A three-level orthogonal discrete wavelet transform analysis filter bank which uses third-order wavelet filters.

4.6.1 Folding Equations for the Original DFG

The multirate DFG in Figure 4.15 has 36 single-rate edges and 6 multirate edges which contain decimators. The number of folded delays on each edge prior to retiming is given in Table 4.4 for the single-rate edges and in Table 4.5 for the multirate edges. These values are computed by using the number of delays on the edges in the DFG and the schedule in Table 4.3 and plugging these values into (4.1) and (4.2).

Table 4.3: Schedule for the three-level orthonormal DWT example. The numbers across the top of the table represent the eight time partitions. An X denotes a null operation, so it is clear that the folded architecture will have 87.5% hardware utilization.

	0	1	2	3	4	5	6	7
Processor M_1	M_{10}	M_{11}	M_{10}	M_{12}	M_{10}	M_{11}	M_{10}	X
Processor M_2	M_{20}	M_{21}	M_{20}	M_{22}	M_{20}	M_{21}	M_{20}	X
Processor M_3	M_{30}	X	M_{30}	M_{31}	M_{30}	M_{32}	M_{30}	M_{31}
Processor M_4	M_{40}	X	M_{40}	M_{41}	M_{40}	M_{42}	M_{40}	M_{41}
Processor M_5	M_{52}	M_{50}	M_{51}	M_{50}	X	M_{50}	M_{51}	M_{50}
Processor M_6	M_{62}	M_{60}	M_{61}	M_{60}	X	M_{60}	M_{61}	M_{60}
Processor A_1	A_{10}	A_{11}	A_{10}	X	A_{10}	A_{11}	A_{10}	A_{12}
Processor A_2	A_{20}	A_{21}	A_{20}	X	A_{20}	A_{21}	A_{20}	A_{22}
Processor A_3	A_{31}	A_{30}	A_{32}	A_{30}	A_{31}	A_{30}	X	A_{30}
Processor A_4	A_{41}	A_{40}	A_{42}	A_{40}	A_{41}	A_{40}	X	A_{40}

4.6.2 Retiming for Folding

There are 36 retiming for folding equations for single-rate edges and 6 for multirate edges. These are given in Table 4.4 for the single-rate edges and in Table 4.5 for the multirate edges. The retiming for folding equations used are (4.4) and (4.5). The columns labeled R_{UV} give the values for the right-hand-side of the inequalities for each edge. Note that we also impose the constraint $r(IN) = 0$. This constraint avoids the possibility of adding new delays at the input which can have the effect of changing the functionality of the circuit as was described in Section 4.4. The columns labeled $r(U)$ and $r(V)$ in Tables 4.4 and 4.5 give a solution to these inequalities.

4.6.3 Folding Equations for the Retimed DFG

Based on the retiming values for the nodes, folding equations can be written for the retimed graph. Because the retiming solutions satisfy all of the retiming-for-folding equations, the folding equations now result in a nonnegative number of delays for each folded edge. The new folding equations are given in Table 4.4 for the single-rate edges

and in Table 4.5 for the multirate edges.

4.6.4 Memory Requirements of the Folded Architecture

The memory in the folded architecture can be found using (4.20). Since the architecture implements the retimed DFG, the number of delays on the folded edges for the retimed graph are used in the expressions in Table 4.2. An important point is that an edge with a decimator can change from set $\mathcal{E}_{U_i}$ to $\mathcal{E}_{U_j}$ as a result of retiming, and this change must be taken into account to get an accurate evaluation of the memory required by the folded architecture. Taking this into account, the minimum number of registers required to implement the folded architecture is 14.

4.6.5 Allocate Data to the Minimum Number of Registers

To keep routing simple, we attempted to localize data within the architecture while still using only 14 registers. For example, we were able to allow only the output samples of multiplier M_1 to occupy registers $R1$ and $R2$ (see Figure 4.16), which avoids routing the outputs of other processors to these two registers. Allocation techniques proposed in [51] were used to allocate the data to the 14 registers.

4.6.6 The Folded Architecture

The folded architecture is shown in Figure 4.16. This architecture uses the theoretical lower limit of 14 registers. Delays denoted as D are internal pipelining delays, while the 14 external registers are labeled R_i. The fact that this architecture has the same functionality as the DFG shown in Figure 4.15 has been verified by simulation using Matlab Simulink.

This is not the only architecture which can be designed for this algorithm using

multirate folding. We have also designed a different architecture, which uses only three multipliers and two adders, using the systematic multirate folding technique proposed in this chapter for the three-level orthogonal discrete wavelet transform which uses 7-th order FIR filters, but this example is not included to save space. This demonstrates that multirate folding can be used to design a broad class of single-rate architectures for multirate DSP applications.

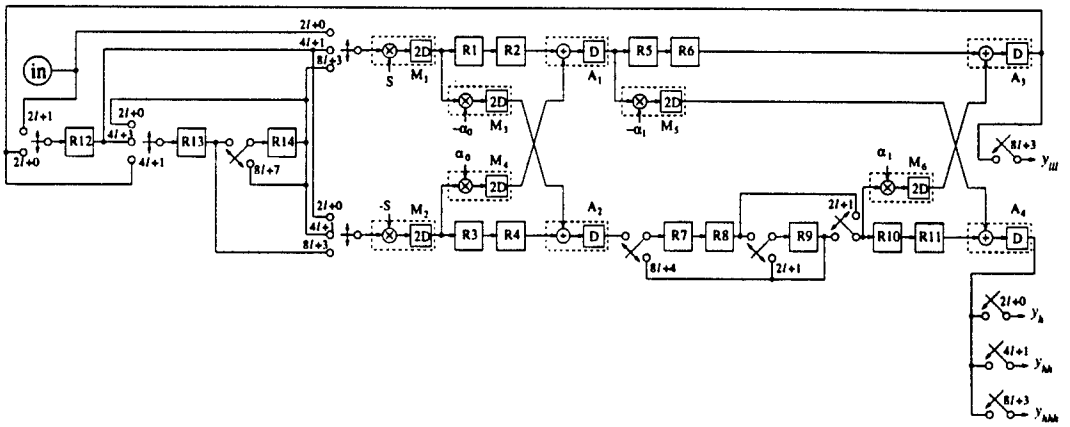


Figure 4.16: Folded architecture for the three-level orthogonal discrete wavelet transform analysis filter bank which uses third-order wavelet filters. If an input to a switch is not labeled, then this input is switched in at all time units not assigned to other inputs of the switch.

4.7 Conclusions

A novel multirate folding transformation has been developed for mapping multirate DSP algorithms to single-rate VLSI architectures. This transformation can be used to synthesize architectures for a wide range of DSP applications which use multirate algorithms, such as signal coding and analysis and adaptive signal processing.

Multirate folding equations were derived for arcs which contain decimators or expanders. In both cases, the folding equation contains single-rate folding as a special case. These folding equations were then used to solve two important related prob-

Table 4.4: Folding and retiming equations for the single-rate edges in the DWT example. The retiming-for-folding equation for edge $U \rightarrow V$ is $r(U) - r(V) \leq R_{UV}$.

U	V	folded delays before retiming	$r(U)$	$r(V)$	R_{UV}	folded delays after retiming
M_{10}	A_{10}	-2	0	2	-1	2
M_{10}	M_{30}	-2	0	1	-1	0
M_{20}	A_{20}	-2	0	2	-1	2
M_{20}	M_{40}	-2	0	1	-1	0
M_{30}	A_{20}	-2	1	2	-1	0
M_{40}	A_{10}	-2	1	2	-1	0
A_{10}	A_{30}	0	2	3	0	2
A_{10}	M_{50}	0	2	2	0	0
A_{20}	A_{40}	2	2	3	1	4
A_{20}	M_{60}	2	2	2	1	2
M_{50}	A_{40}	-2	2	3	-1	0
M_{60}	A_{30}	-2	2	3	-1	0
M_{11}	A_{11}	-2	2	3	-1	2
M_{11}	M_{31}	0	2	2	0	0
M_{21}	A_{21}	-2	2	3	-1	2
M_{21}	M_{41}	0	2	2	0	0
M_{31}	A_{21}	-4	2	3	-1	0
M_{41}	A_{11}	-4	2	3	-1	0
A_{11}	A_{31}	-2	3	4	-1	2
A_{11}	M_{51}	0	3	3	0	0
A_{21}	A_{41}	2	3	4	0	6
A_{21}	M_{61}	4	3	3	1	4
M_{51}	A_{41}	-4	3	4	-1	0
M_{61}	A_{31}	-4	3	4	-1	0
M_{12}	A_{12}	2	2	2	0	2
M_{12}	M_{32}	0	2	2	0	0
M_{22}	A_{22}	2	2	2	0	2
M_{22}	M_{42}	0	2	2	0	0
M_{32}	A_{22}	0	2	2	0	0
M_{42}	A_{12}	0	2	2	0	0
A_{12}	A_{32}	-6	2	3	-1	2
A_{12}	M_{52}	-8	2	3	-1	0
A_{22}	A_{42}	2	2	3	0	10
A_{22}	M_{62}	0	2	3	0	8
M_{52}	A_{42}	0	3	3	0	0
M_{62}	A_{32}	0	3	3	0	0

Table 4.5: Folding and retiming equations for the multirate edges in the DWT example. The retiming-for-folding equation for edge $U \rightarrow V$ is $r(U) - 2r(V) \leq R_{UV}$.

U	V	folded delays before retiming	$r(U)$	$r(V)$	R_{UV}	folded delays after retiming
IN	M_{10}	0	0	0	0	0
IN	M_{20}	1	0	0	1	1
A_{30}	M_{11}	-1	3	2	-1	1
A_{30}	M_{21}	1	3	2	0	3
A_{31}	M_{12}	2	4	2	0	2
A_{31}	M_{22}	6	4	2	1	6

lems, namely, memory minimization in folded architectures and retiming for folding. By deriving the multirate folding equations and solving these related problems, we have formalized several crucial steps used in mapping multirate DSP algorithms to efficient VLSI architectures.

A detailed design example of a three-level discrete wavelet transform analysis filter bank was given. This example demonstrated how the multirate folding equations, along with retiming for folding and memory minimization, can be used to design single-rate architectures for multirate algorithms. Multirate folding can be used to design architectures for a wide variety of filter banks as described in [36].

Chapter 5

Two-Dimensional Retiming

5.1 Introduction

Retiming [27] is a technique used to move delay elements around in a circuit without changing its functionality. One effect of changing the locations of the delays is that combinational rippling can be reduced, allowing the the circuit to be clocked at a higher rate. Reducing combinational rippling also decreases the dynamic power dissipation in the circuit [48] and allows the circuit to be operated with a lower supply voltage, both of which lead to low power implementations [67]. Another effect of changing the locations of delays is that the number of delay elements required can be reduced, resulting in area-efficient implementations. In addition to retiming for high speed, low power, and low area implementations, retiming is also an important step in scheduling for high-level synthesis [11] -[38]. All of these applications of retiming have been studied for circuits which operate on one-dimensional signals, such as digital audio.

Two-dimensional retiming [33, 34] is used to retime data-flow graphs (DFGs) which operate on two-dimensional signals such as images. As digital image processing becomes more popular in multimedia applications, the need for high speed, low area, and low power implementations of multidimensional digital signal processing (DSP) algorithms

increases. Like one-dimensional retiming [27], two-dimensional retiming can be used to increase the sample rate, reduce the area, and reduce the power consumed by a synchronous circuit.

Techniques for reducing the execution times of 2-D DSP algorithms have been considered in the past. One way to speed up these algorithms is to process many iterations concurrently, and it has been shown that this is often possible if the 2-D data are not processed in line-by-line or column-by-column order, but rather are processed diagonally [69, 70]. This technique requires an increase in the number of arithmetic units. Another way to speed up these algorithms is to reduce the sample period using 2-D retiming techniques [33, 34]. This technique does not require an increase in the number of arithmetic units; however, as we show in this chapter, the algorithm for 2-D retiming in [34] often results in an implementation which requires significantly more memory than is actually needed. Since the area consumed by the implementation of a 2-D DSP algorithm can be dominated by memory requirements [71], it is important to keep the memory requirements as small as possible. The algorithm for 2-D retiming in [33] is not very flexible because it is only compatible with some very specific processing orders of the data.

In this chapter, we present two techniques for retiming two-dimensional data-flow graphs (2DFGs). Each of these techniques minimizes the amount of memory required to implement the 2DFG under a clock period constraint. The first technique, called *ILP 2-D retiming*, is based on an integer linear programming (ILP) formulation which considers the 2-D retiming formulation as a whole. While this technique gives excellent results, it has slow convergence for large 2DFGs. The second technique, called *orthogonal 2-D retiming*, is formulated by breaking ILP 2-D retiming into two linear programming problems, where each problem can be solved in polynomial time. The downfall of orthogonal 2-D retiming is that the results of the two linear programming problems can sometimes

be incompatible. A variation of orthogonal 2-D retiming called *integer orthogonal 2-D retiming* is also based on a linear programming formulation, and this technique solves the incompatibility problem which may be encountered using orthogonal 2-D retiming. The techniques presented in this chapter result in retimed 2DFGs which require less memory than than the technique in [34] and are compatible with considerably more processing orders of the data than the technique described in [33].

This chapter is organized as follows. Section 5.2 describes some specifics of two-dimensional data processing. Section 5.3 contains the ILP 2-D retiming formulation. Orthogonal 2-D retiming and integer orthogonal 2-D retiming are presented in Sections 5.4 and 5.5, respectively. Comparisons with previous work are given in Section 5.6 and our conclusions are in Section 5.7.

5.2 Processing Two-Dimensional Data Sets

A two-dimensional DSP algorithm can be represented using a two-dimensional data-flow graph (2DFG). A 2DFG $G =< V, E, \mathbf{w}, d >$ is a node-weighted and edge-weighted graph such that

- V is the set of vertices (nodes) in G. The nodes represent computations.
- E is the set of edges in G. The edges represent communication between the nodes.
- $\mathbf{w}(e)$ is a 2×1 vector representing the dependency on edge e.
- $d(v)$ is a nonnegative scalar representing the computation time of node v.

As an example, the 2DFG in Figure 5.1 describes the computation $y(n_1, n_2) = b + ax(n_1 + 1, n_2 - 1)$. An *iteration* is the execution of each node in the 2DFG exactly once.

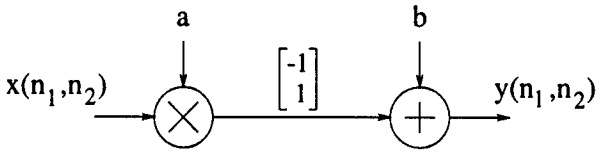


Figure 5.1: A 2DFG which describes the computation $y(n_1, n_2) = b + ax(n_1 + 1, n_2 - 1)$.

5.2.1 Overview of Two-Dimensional Retiming

The 1-D retiming equation given in [27] for the edge $u \xrightarrow{e} v$ in a 1-D DFG is given by

$$w_r(e) = w(e) + r(v) - r(u),$$

where $w(e)$ and $w_r(e)$ are the numbers of delays on e before and after retiming, respectively, and $r(u)$ and $r(v)$ are the retiming values of nodes u and v, respectively. The 2-D retiming equation for the edge $u \xrightarrow{e} v$ in a 2DFG is given by

$$\mathbf{w}_r(e) = \mathbf{w}(e) + \mathbf{r}(v) - \mathbf{r}(u), \tag{5.1}$$

where $\mathbf{w}(e)$ and $\mathbf{w}_r(e)$ are the 2×1 dependence vectors on e before and after retiming, respectively, and $\mathbf{r}(u)$ and $\mathbf{r}(v)$ are the 2×1 retiming vectors of nodes u and v, respectively.

A 1-D retiming r is said to be *legal* if $w_r(e) \geq 0$ for all $e \in E$. The conditions for a legal 2-D retiming are derived in Section 5.3.1.

5.2.2 Types of Parallelism Available in 2-D Signal Processing

There are two types of parallelism available in 2-D signal processing. The first type of parallelism is *inter-iteration parallelism* which can be achieved by increasing the amount of hardware so that the multiple iterations can be executed concurrently. For example, consider the 2DFG in Figure 5.2(a) which implements $y(n_1, n_2) = ay(n_1 - 1, n_2) + by(n_1, n_2 - 1) + x(n_1, n_2)$. Assume that this 2DFG is used to process a 3×3 data set.

Table 5.1: Four possible execution orders for the DFG in Figure 5.2(a) assuming a 3×3 data set.

	row-by-row serial	column-by-column serial	diagonal serial	parallel
Step 1	$y(0,0)$	$y(0,0)$	$y(0,0)$	$y(0,0)$
Step 2	$y(1,0)$	$y(0,1)$	$y(1,0)$	$y(0,1), y(1,0)$
Step 3	$y(2,0)$	$y(0,2)$	$y(0,1)$	$y(0,2), y(1,1), y(2,0)$
Step 4	$y(0,1)$	$y(1,0)$	$y(2,0)$	$y(1,2), y(2,1)$
Step 5	$y(1,1)$	$y(1,1)$	$y(1,1)$	$y(2,2)$
Step 6	$y(2,1)$	$y(1,2)$	$y(0,2)$	–
Step 7	$y(0,2)$	$y(2,0)$	$y(2,1)$	–
Step 8	$y(1,2)$	$y(2,1)$	$y(1,2)$	–
Step 9	$y(2,2)$	$y(2,2)$	$y(2,2)$	–

The output values $y(n_1, n_2)$ are dependent on one another as shown in Figure 5.2(b), where, e.g., the arrow from $y(1,0)$ to $y(1,1)$ indicates that $y(1,0)$ must be computed before $y(1,1)$ can be computed. Four possible execution orders are given in Table 5.1.

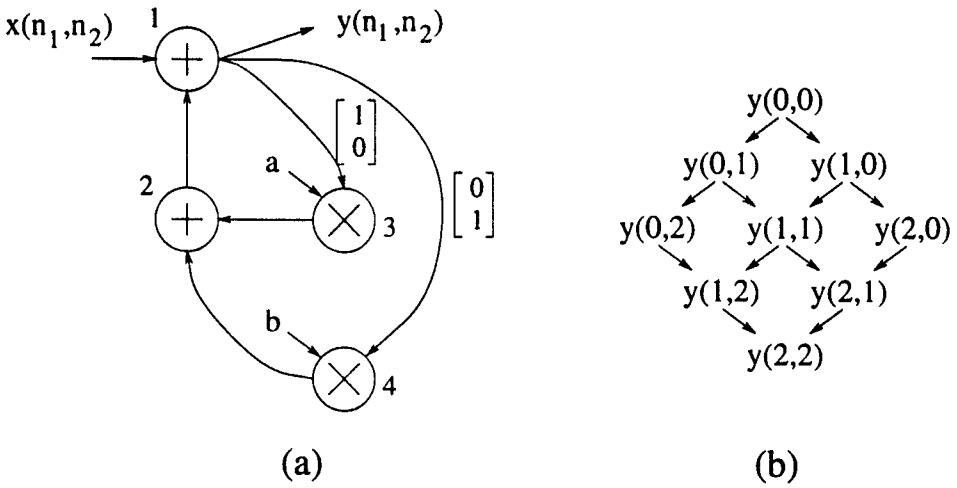


Figure 5.2: (a) A 2DFG which describes the computation $y(n_1, n_2) = ay(n_1 - 1, n_2) + by(n_1, n_2 - 1) + x(n)$. (b) The dependencies for this 2DFG assuming it operates on a 3×3 data set.

While the three serial execution orders require a single hardware module and 9 time steps to execute, the parallel execution order requires 3 hardware modules and only five time steps to execute, where a hardware module is capable of executing one iteration in

one time step. The parallel execution order uses inter-iteration parallelism to speed-up the execution of the 2-D signal processing algorithm.

The second type of parallelism is *inter-operation parallelism.* This involves retiming the 2DFG so operations can be executed in parallel, resulting in a shorter clock period. For the 2DFG in Figure 5.2(a), assume addition and multiplication require 1 and 2 time units, respectively. The minimum clock period for this 2DFG is 4 time units because there is a path through two adders and one multiplier (e.g., through nodes 4, 2, and 1) which has no delays. As a result, the time required to process the 3×3 data set using a serial processing order is $(4)(9) = 36$ time units. The 2DFG in Figure 5.2(a) can be retimed as shown in Figure 5.3 assuming $\mathbf{r}(1) = [\ 0 \quad 0\ ]^T$, $\mathbf{r}(2) = [\ 0 \quad 0\ ]^T$, $\mathbf{r}(3) = [\ -2 \quad 1\ ]^T$, and $\mathbf{r}(4) = [\ -1 \quad 0\ ]^T$. This retimed 2DFG has a minimum clock period of 2 time units because the longest path with no delays is through a multiplier or two adders. The time required to process the 3×3 data set using the diagonal serial processing order is now $(2)(9) = 18$ time units, so 2-D retiming has allowed us to speed up the processing by a factor of 2.

The reason that 2-D retiming allows the circuit to be clocked faster is because operations in the retimed circuit can be executed in parallel. Table 5.2 shows some possible execution times for the nodes in the unretimed 2DFG (Figure 5.2(a)) and the retimed 2DFG (Figure 5.3). Since multiplication and addition in the retimed 2DFG can be performed in parallel rather than sequentially, 2-D retiming allows for an implementation where operations are executed in parallel, hence the name *inter-operation parallelism.* The remainder of this chapter assumes that a 2-D data set is processed using a serial processing order, and we focus on exploiting inter-operation parallelism using 2-D retiming.

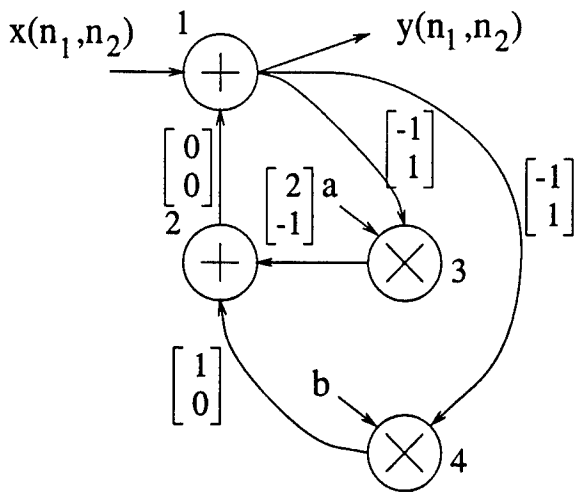


Figure 5.3: A retimed version of the 2DFG in Figure 5.2(a).

Table 5.2: Possible execution times for the unretimed 2DFG in Figure 5.2(a) and the retimed 2DFG in Figure 5.3 assuming that addition and multiplication require 1 and 2 units of time, respectively. The unretimed 2DFG does not allow addition and multiplication to be executed in parallel, while the retimed 2DFG does allow addition and multiplication to be executed in parallel.

	unretimed 2DFG				retimed 2DFG			
time	node 1	node 2	node 3	node 4	node 1	node 2	node 3	node 4
0			*	*		*	*	*
1			*	*	*		*	*
2		*				*	*	*
3	*				*		*	*

5.2.3 Processing Order

A two-dimensional DSP algorithm can often be executed using several processing orders. This was demonstrated in the previous section where three serial processing orders were given in Table 5.1 for the 2DFG in Figure 5.2(a). A linear processing order is specified using a scanning vector $\mathbf{s} = [\ s_1 \quad s_2\ ]^T$ and an access vector $\mathbf{a} = [\ a_1 \quad a_2\ ]^T$. Lines orthogonal to the scanning vector are called *access lines*, and sample (n_1, n_2) on access line k satisfies $n_1 s_1 + n_2 s_2 = k$. The processing order is such that, for $k_1 < k_2$, all samples on access line k_1 are processed before the samples on access line k_2. The access vector,

which is orthogonal to the scanning vector ($\mathbf{s} \cdot \mathbf{a} = 0$), defines the order in which samples are processed on the access lines, such that sample $\mathbf{n} + \mathbf{a}$ is processed immediately following sample $\mathbf{n}$. Lines orthogonal to the access vector are called *scanning lines*, and sample (n_1, n_2) on scanning line k satisfies $n_1 a_1 + n_2 a_2 = k$. As an example, the processing order in Figure 5.4 is described by $\mathbf{s} = [\ 1 \quad 1\ ]^T$ and $\mathbf{a} = [\ \text{-}1 \quad 1\ ]^T$, and sample $(2, 4)$ is on access line 6 and scanning line 2. In addition to linear processing orders, nonlinear processing orders such as the Dovetail scan [72] also exist; however, this chapter considers only linear processing orders.

5.3 An Integer Linear Programming Formulation of 2-D Retiming

In this section we formulate the *ILP 2-D retiming* technique which considers causality, the desired clock period, and the memory cost of the 2-D retiming solution.

5.3.1 Causality in 2-D Data Processing

A dependency $w(e)$ in a 1-D DFG must represent a causal relationship. If the edge $u \xrightarrow{e} v$ has a negative number of delays, this indicates that node v is consuming data before node u has produced the data, and this is not practical from an implementation point of view. Causality restricts the number of delays on an edge in a 1-D DFG to be nonnegative, which can be written as $w(e) \geq 0$ for all $e \in E$. The expression $w(e) \geq 0$ for all $e \in E$ can be viewed as the condition for the compatibility between the dependencies and the order in which the data is processed (which is dictated by time).

In 2DFGs, where the processing order is specified by $\mathbf{s}$ and $\mathbf{a}$, there are two conditions for the compatibility between the dependencies and the processing order. These two conditions are the 2-D causality constraints. The first causality constraint states that

a dependency $\mathbf{w}(e)$ on the edge $u \xrightarrow{e} v$ cannot point from access line k_2 to access line k_1 for $k_1 < k_2$ because this would indicate that the data produced when access line k_2 is processed is consumed when access line k_1 is processed, and this violates causality because access line k_2 is processed after access line k_1. Mathematically, this causality constraint can be written as

Causality Constraint 5.1 *For all* $e \in E$, $\mathbf{s} \cdot \mathbf{w}(e) \geq 0$ *must hold.*

The second causality constraint states that if the dependency $\mathbf{w}(e)$ lies in the same direction as the access lines, then the dependency cannot point in the opposite direction as the access vector because this would mean that the dependency points to the opposite direction of processing of data. This can be expressed as

Causality Constraint 5.2 *For all* $e \in E$ *such that* $\mathbf{s} \cdot \mathbf{w}(e) = 0$, $\mathbf{a} \cdot \mathbf{w}(e) \geq 0$ *must hold.*

Example 5.1 *For* $\mathbf{s} = [\ 1 \quad 1\ ]^T$ *and* $\mathbf{a} = [\ -1 \quad 1\ ]^T$, *Figure 5.4 shows how four different dependencies would affect the sample at the* $(2,3)$ *location. The dependency* $\mathbf{w}(e) = [\ 0 \quad -1\ ]^T$ *represents a non-causal relationship because the value computed when sample* $(2,4)$ *is processed affects the value at sample* $(2,3)$, *but sample* $(2,4)$ *is processed after* $(2,3)$. *This dependency violates the first causality constraint because* $\mathbf{s} \cdot \mathbf{w}(e) = -1$. *The dependency* $\mathbf{w}(e) = [\ 0 \quad 1\ ]^T$ *represents a causal relationship because the value computed when sample* $(2,2)$ *is processed affects the value at sample* $(2,3)$, *and sample* $(2,2)$ *is processed before* $(2,3)$. *This dependency satisfies the first causality constraint because* $\mathbf{s} \cdot \mathbf{w}(e) = 1$. *The dependency* $\mathbf{w}(e) = [\ 1 \quad -1\ ]^T$ *represents a non-causal relationship because the value computed when sample* $(1,4)$ *is processed affects the value at sample* $(2,3)$, *but sample* $(1,4)$ *is processed after* $(2,3)$. *This dependency violates the second causality constraint because* $\mathbf{a} \cdot \mathbf{w}(e) = -2$ *and* $\mathbf{s} \cdot \mathbf{w}(e) = 0$. *The depen-*

dency $\mathbf{w}(e) = [\ -1 \quad 1\ ]^T$ *represents a causal relationship because the value computed when sample* $(3,2)$ *is processed affects the value at sample* $(2,3)$*, and sample* $(3,2)$ *is processed before* $(2,3)$*. This dependency satisfies the second causality constraint because* $\mathbf{a} \cdot \mathbf{w}(e) = 2$ *and* $\mathbf{s} \cdot \mathbf{w}(e) = 0$. □

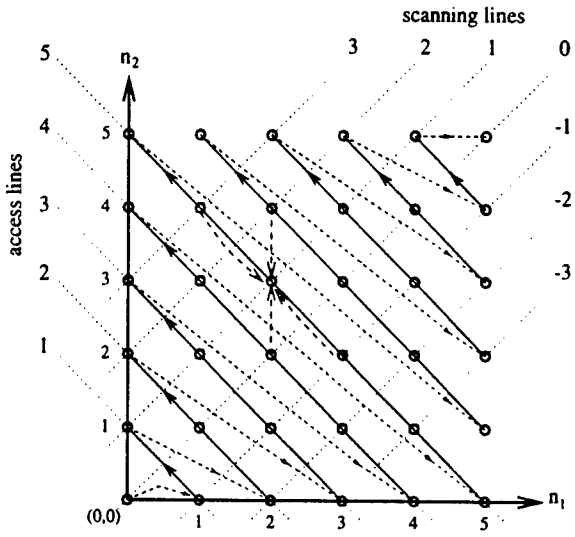


Figure 5.4: The effect of four dependencies on sample $(2,3)$. Processing starts at sample $(0,0)$.

Let H_{max} be the maximum number of samples on any access line. Then the length of the longest access line is $(H_{max} - 1)(\mathbf{a} \cdot \mathbf{a})$. In a practical situation, the length of each dependence vector is not greater than the length of the longest access line, and this implies that the projection of a dependence vector onto the access vector obeys

$$H_{max}(\mathbf{a} \cdot \mathbf{a}) > |\mathbf{a} \cdot \mathbf{w}(e)|. \tag{5.2}$$

This inequality is used in the following theorem to combine the two causality constraints into a single constraint.

Theorem 5.1 *Let (5.2) hold for all* $e \in E$*. Then*

$$H_{max}(\mathbf{a} \cdot \mathbf{a})(\mathbf{s} \cdot \mathbf{w}(e)) + \mathbf{a} \cdot \mathbf{w}(e) \geq 0 \tag{5.3}$$

if and only if the following hold:

1. $\mathbf{s} \cdot \mathbf{w}(e) \geq 0$, *and*

2. $\mathbf{a} \cdot \mathbf{w}(e) \geq 0$ *if* $\mathbf{s} \cdot \mathbf{w}(e) = 0$.

Proof: In the first part of the proof, we show that (5.3) implies

1. $\mathbf{s} \cdot \mathbf{w}(e) \geq 0$, and

2. $\mathbf{a} \cdot \mathbf{w}(e) \geq 0$ if $\mathbf{s} \cdot \mathbf{w}(e) = 0$.

The expression in (5.3) can be written as

$$\mathbf{s} \cdot \mathbf{w}(e) \geq \frac{-(\mathbf{a} \cdot \mathbf{w}(e))}{H_{max}(\mathbf{a} \cdot \mathbf{a})}.$$

Using (5.2), this can be written as $\mathbf{s} \cdot \mathbf{w}(e) > -1$. Since $\mathbf{s} \cdot \mathbf{w}(e)$ is an integer, this implies $\mathbf{s} \cdot \mathbf{w}(e) \geq 0$. When $\mathbf{s} \cdot \mathbf{w}(e) = 0$, the expression in (5.3) simplifies to $\mathbf{a} \cdot \mathbf{w}(e) \geq 0$.

In the second part of the proof, we show that

1. $\mathbf{s} \cdot \mathbf{w}(e) \geq 0$, and

2. $\mathbf{a} \cdot \mathbf{w}(e) \geq 0$ if $\mathbf{s} \cdot \mathbf{w}(e) = 0$

imply (5.3). If $\mathbf{s} \cdot \mathbf{w}(e) \geq 1$, then (5.3) holds because (5.2) states that $\mathbf{a} \cdot \mathbf{w}(e) \geq -H_{max}(\mathbf{a} \cdot \mathbf{a})$. If $\mathbf{s} \cdot \mathbf{w}(e) = 0$, then (5.3) holds because $\mathbf{a} \cdot \mathbf{w}(e) \geq 0$. □

If we let

$$F(\mathbf{x}) = H_{max}(\mathbf{a} \cdot \mathbf{a})(\mathbf{s} \cdot \mathbf{x}) + \mathbf{a} \cdot \mathbf{x},$$

then causality can be written as $F(\mathbf{w}(e)) \geq 0$ for all $e \in E$. This definition of $F(\mathbf{x})$ is used throughout the remainder of the chapter. For a retimed 2DFG G_r, causality can be written as $F(\mathbf{w}_r(e)) \geq 0$ for all $e \in E$. A 2-D retiming $\mathbf{r}$ from G to G_r is *legal* if $F(\mathbf{w}(e)) \geq 0$ for all $e \in E$.

5.3.2 The Clock Period Constraints

In this section we develop the constraints which can be used to specify a desired clock period for the retimed 2DFG. Let $p = v_0 \stackrel{e_0}{\rightarrow} v_1 \stackrel{e_1}{\rightarrow} \cdots \stackrel{e_{k-1}}{\rightarrow} v_k$ be a path in the 2DFG. The delay of the path is $d(p) = \sum_{i=0}^{k} d(v_i)$ and the dependency of the path is $\mathbf{w}(p) = \sum_{i=0}^{k-1} \mathbf{w}(e_i)$. The clock period $\Phi(G)$ is defined to be the maximum propagation delay through which any signal must ripple between clock cycles. Mathematically,

$$\Phi(G) = \max\{d(p) : \mathbf{w}(p) = \mathbf{0}\}.$$

The derivations in this section follow the derivations in [27].

Let

$$W(u,v) = \min\{F(\mathbf{w}(p)) : u \stackrel{p}{\rightarrow} v\}$$

and

$$D(u,v) = \max\{d(p) : u \stackrel{p}{\rightarrow} v \quad \text{and} \quad F(\mathbf{w}(p)) = W(u,v)\}.$$

Lemma 5.2 *Let G be a 2DFG, and let c be any positive real number. The following are equivalent.*

5.2.1 $\Phi(G) \leq c$.

5.2.2 For all vertices u and v in V, if $D(u,v) > c$, then $W(u,v) \geq \mathbf{a} \cdot \mathbf{a}$.

Proof: (5.2.1 $\Rightarrow$ 5.2.2): Suppose $\Phi(G) \leq c$ and let u and v be vertices such that $D(u,v) > c$. Assume that $W(u,v) < \mathbf{a} \cdot \mathbf{a}$. If all edges in G are causal, then $W(u,v) = 0$, so there exists a path $u \stackrel{p}{\rightarrow} v$ with propagation delay $d(p) = D(u,v) > c$ and $F(\mathbf{w}(p)) = W(u,v) = 0$, which implies $\mathbf{w}(p) = 0$ and $\Phi(G) > c$. Contradiction.

(5.2.2 $\Rightarrow$ 5.2.1): Suppose 5.2.2 holds and let $u \stackrel{p}{\rightarrow} v$ be any path in G such that $F(\mathbf{w}(p)) = 0$. Then we have $W(u,v) = F(\mathbf{w}(p)) = 0$, which implies $d(p) \leq D(u,v) \leq c$

(this is the contrapositive of "if $D(u,v) > c$ then $W(u,v) \geq \mathbf{a} \cdot \mathbf{a}$"). This implies 5.2.1. □

A critical path is any path $u \xrightarrow{p} v$ with $F(\mathbf{w}(p)) = W(u,v)$. Assume that $\mathbf{r}$ is a 2-D retiming that satisfies the causality constraints for a given processing order specified by $\mathbf{s}$ and $\mathbf{a}$. Let $W_r(u,v)$ and $D_r(u,v)$ have the same definitions on the retimed graph G_r as $W(u,v)$ and $D(u,v)$ have on G. The following can be proven using techniques similar to those used for the 1-D case [27].

- $W_r(u,v) = W(u,v) + F(\mathbf{r}(v) - \mathbf{r}(u))$.
- a path p is a critical path of G_r if and only if it is a critical path of G.
- $D_r(u,v) = D(u,v)$ for all connected $u, v \in V$
- the clock period $\Phi(G_r)$ is equal to $D(u,v)$ for some $u, v \in V$.

Using these results, we can prove the following.

Theorem 5.3 *Let c be an arbitrary real number and let $\mathbf{s}$ and $\mathbf{a}$ be orthogonal vectors which specify a linear processing order. Then $\mathbf{r}$ is a legal retiming such that $\Phi(G_r) \leq c$ if and only if*

5.3.1 $F(\mathbf{r}(u) - \mathbf{r}(v)) \leq F(\mathbf{w}(e))$ for every edge $u \xrightarrow{e} v$ of G, and

5.3.2 $F(\mathbf{r}(u) - \mathbf{r}(v)) \leq W(u,v) - \mathbf{a} \cdot \mathbf{a}$ for all vertices $u, v \in V$ such that $D(u,v) > c$.

Proof: The retiming is legal if and only if 5.3.1 holds. If $\mathbf{r}$ is indeed a legal retiming of G, then by Lemma 5.2 the retimed circuit G_r has clock period $\Phi(G_r) \leq c$ under the condition that $W_r(u,v) \geq \mathbf{a} \cdot \mathbf{a}$ for all vertices $u, v \in V$ such that $D_r(u,v) > c$. Since we know $D_r(u,v) = D(u,v)$ and $W_r(u,v) = W(u,v) + F(\mathbf{r}(v) - \mathbf{r}(u))$, G_r has $\Phi(G_r) \leq c$

under the condition that $W(u,v) \geq -F(\mathbf{r}(v) - \mathbf{r}(u)) + \mathbf{a} \cdot \mathbf{a}$ for all $u, v \in V$ such that $D(u,v) > c$. Since $F(\mathbf{r}(v) - \mathbf{r}(u)) = -F(\mathbf{r}(u) - \mathbf{r}(v))$, this is equivalent to 5.3.2. □

5.3.3 The Memory Cost

For the ILP formulation to be complete, it requires a linear approximation of the number of registers required to implement the retimed circuit. A linear approximation for the number of registers required to implement the dependency $\mathbf{w}(e)$ should consider the number of access lines and scanning lines crossed by the dependency. The number of access lines crossed is $\mathbf{s} \cdot \mathbf{w}(e)$, and the maximum number of samples in an access line is H_{max}, so an upper bound on the number of registers required to store $\mathbf{s} \cdot \mathbf{w}(e)$ access lines is $H_{max}(\mathbf{s} \cdot \mathbf{w}(e))$. The number of scanning lines crossed by $\mathbf{w}(e)$ is $\mathbf{a} \cdot \mathbf{w}(e)$, and one register is required for $\mathbf{a} \cdot \mathbf{a}$ scanning lines that are crossed (to see this, consider that the dependency corresponding to a single sample delay is $\mathbf{w}(e) = \mathbf{a}$); so an estimate for the number of registers required due to scanning lines that are crossed is $(\mathbf{a} \cdot \mathbf{w}(e))/(\mathbf{a} \cdot \mathbf{a})$. The linear approximation for the total number of registers required to implement the dependency $w(e)$ is $H_{max}(\mathbf{s} \cdot \mathbf{w}(e)) + (\mathbf{a} \cdot \mathbf{w}(e))/(\mathbf{a} \cdot \mathbf{a})$, which can be written as $F(\mathbf{w}(e))/(\mathbf{a} \cdot \mathbf{a})$.

If a node has more than one output edge carrying the same signal (such a node is often called a *fanout node*), the number of registers required to implement these edges is the maximum number of registers on any one of them [21]. This is shown in Figure 5.5 for the 1-D case, where the naive implementation in Figure 5.5(a) uses $1 + 3 + 7 = 11$ registers while the efficient implementation in Figure 5.5(b) uses $\max(1, 3, 7) = 7$ registers. Using this concept, the number of registers required to implement the output edges of node v is estimated to be

$$R_v = \max_{v \xrightarrow{e} ?} \{F(\mathbf{w}_r(e))/(\mathbf{a} \cdot \mathbf{a})\}.$$

The cost function can be minimized by using $COST = \sum_{v \in V} R_v$ where $R_v \geq F(\mathbf{w}_r(e))$ for all edges $v \xrightarrow{e} ?$. Note that this cost represents the number of memory locations scaled by a constant scale factor $(\mathbf{a} \cdot \mathbf{a})$.

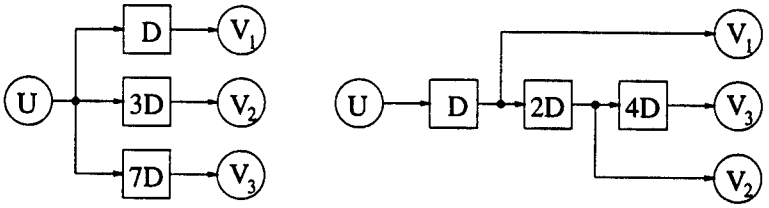


Figure 5.5: (a) Fanout implementation using $1 + 3 + 7 = 11$ registers. (b) Fanout implementation using $\max(1, 3, 7) = 7$ registers.

5.3.4 The Complete ILP 2-D Retiming Formulation

Theorem 5.3 specifies the conditions for a retiming to be legal and satisfy a given clock period constraint. Combining this with the cost function, the complete ILP formulation of 2-D retiming is: Minimize $COST = \sum_{v \in V} R_v$ under the constraints

1. $R_v \geq F(\mathbf{w}_r(e))$ for all edges $v \xrightarrow{e} ?$ and all $v \in V$ (fanout constraint).
2. $F(\mathbf{r}(u) - \mathbf{r}(v)) \leq F(\mathbf{w}(e))$ for every edge $u \xrightarrow{e} v$ of G (causality constraint).
3. $F(\mathbf{r}(u) - \mathbf{r}(v)) \leq W(u, v) - \mathbf{a} \cdot \mathbf{a}$ for all vertices $u, v \in V$ such that $D(u, v) > c$ (clock period constraint).

Example 5.2 *Consider the 2DFG in Figure 5.6(a). Assume that the computation time for each node is* 1 *time unit. The goal is to retime this 2DFG to minimize the memory while achieving a clock period of* $\Phi(G_r) = 1$ *assuming an* 8×8 *data set and a processing order specified by* $\mathbf{s} = [\ 1 \quad 2\ ]^T$ *and* $\mathbf{a} = [\ -2 \quad 1\ ]^T$. *The maximum number of samples on an access line is* $H_{max} = 4$ *and* $\mathbf{a} \cdot \mathbf{a} = 5$, *so* $F(\mathbf{x}) = 20(\mathbf{s} \cdot \mathbf{x}) + \mathbf{a} \cdot \mathbf{x}$. *The ILP formulation*

is to minimize $COST = R_1 + R_2 + R_3 + R_4$ *subject to the fanout constraints, the causality constraints, and the clock period constraints. The five fanout constraints are*

$$\begin{aligned} R_1 &\geq 0 + F(\mathbf{r}(2) - \mathbf{r}(1)) \\ R_1 &\geq 0 + F(\mathbf{r}(3) - \mathbf{r}(1)) \\ R_2 &\geq 23 + F(\mathbf{r}(4) - \mathbf{r}(2)) \\ R_3 &\geq 59 + F(\mathbf{r}(4) - \mathbf{r}(3)) \\ R_4 &\geq 0 + F(\mathbf{r}(1) - \mathbf{r}(4)). \end{aligned}$$

The five causality constraints are

$$\begin{aligned} F(\mathbf{r}(1) - \mathbf{r}(2)) &\leq 0 \\ F(\mathbf{r}(1) - \mathbf{r}(3)) &\leq 0 \\ F(\mathbf{r}(2) - \mathbf{r}(4)) &\leq 23 \\ F(\mathbf{r}(3) - \mathbf{r}(4)) &\leq 59 \\ F(\mathbf{r}(4) - \mathbf{r}(1)) &\leq 0. \end{aligned}$$

The values of $W(u,v)$ *and* $D(u,v)$ *are given in Table 5.3, and based on these values the twelve clock period constraints are*

$$\begin{aligned} F(\mathbf{r}(1) - \mathbf{r}(2)) &\leq -5 \\ F(\mathbf{r}(1) - \mathbf{r}(3)) &\leq -5 \\ F(\mathbf{r}(1) - \mathbf{r}(4)) &\leq 18 \\ F(\mathbf{r}(2) - \mathbf{r}(1)) &\leq 18 \\ F(\mathbf{r}(2) - \mathbf{r}(3)) &\leq 18 \\ F(\mathbf{r}(2) - \mathbf{r}(4)) &\leq 18 \\ F(\mathbf{r}(3) - \mathbf{r}(1)) &\leq 54 \\ F(\mathbf{r}(3) - \mathbf{r}(2)) &\leq 54 \\ F(\mathbf{r}(3) - \mathbf{r}(4)) &\leq 54 \\ F(\mathbf{r}(4) - \mathbf{r}(1)) &\leq -5 \\ F(\mathbf{r}(4) - \mathbf{r}(2)) &\leq -5 \\ F(\mathbf{r}(4) - \mathbf{r}(3)) &\leq -5. \end{aligned}$$

The retiming solution, found using the ILP solver GAMS [63], is $\mathbf{r}(1) = [\ 0 \quad 1\ ]^T$, $\mathbf{r}(2) = [\ 3 \quad 0\ ]^T$, $\mathbf{r}(3) = [\ 3 \quad 0\ ]^T$, *and* $\mathbf{r}(4) = [\ 2 \quad 0\ ]^T$. *The values of* R_1, R_2, R_3, *and* R_4 *are* 13, 5, 41, *and* 5, *respectively, and the total cost is* $COST = 64$. *The retimed 2DFG is shown in Figure 5.6(b).*

A downfall of the ILP 2-D retiming is its slow convergence time. From our experiences, we have found that the ILP solver can take several minutes to find an optimal solution for 2DFGs with as few as 12 nodes. The linear programming formulation in the next

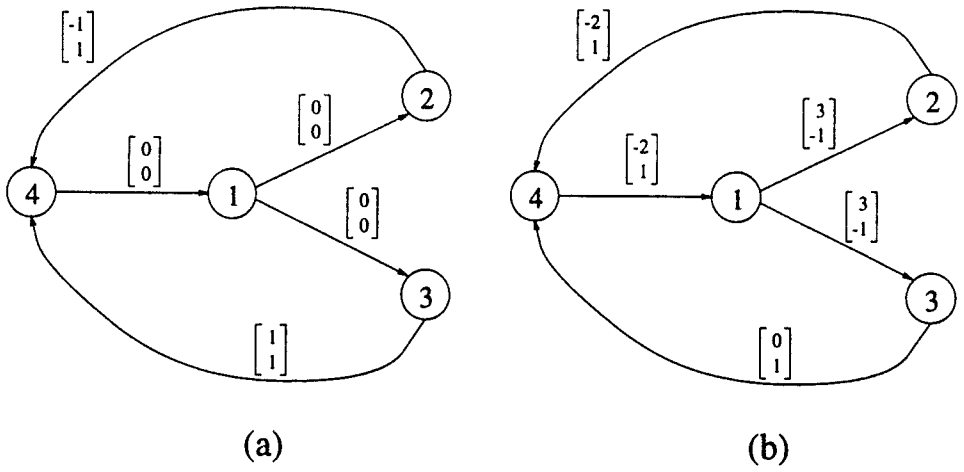


Figure 5.6: The (a) unretimed and (b) retimed 2DFGs referred to in Example 5.2.

Table 5.3: The values of $W(u,v)$ and $D(u,v)$ for Example 5.2.

$W(u,v)$	1	2	3	4
1	0	0	0	23
2	23	0	23	23
3	59	59	0	59
4	0	0	0	0

$D(u,v)$	1	2	3	4
1	1	2	2	3
2	3	1	4	2
3	3	4	1	2
4	2	3	3	1

section can be solved in polynomial time, resulting in significantly faster solution times than ILP 2-D retiming.

5.4 Orthogonal 2-D Retiming

Orthogonal two-dimensional retiming partitions the 2-D retiming problem into two 1-D retiming problems. These 1-D retiming problems, which we call s-retiming and a-retiming, can be solved in polynomial time using techniques similar to those introduced in [27]. By partitioning the 2-D retiming problem into two 1-D retiming problems, some quality of the final solution may be sacrificed because the final solution is no longer guaranteed to be globally optimal; however, our experience has shown that orthogonal 2-D retiming finds solutions that are comparable to the ILP solutions, and these solutions

are found in much faster CPU times than the ILP solutions.

Simply stated, orthogonal 2-D retiming is performed by first performing s-retiming and then performing a-retiming, where these two tasks are specified below:

- s-retiming: Project the 2-D retiming problem onto the **s**-vector and solve this 1-D retiming problem to find the values of $\mathbf{s} \cdot \mathbf{w}_r(e)$ for $e \in E$.
- a-retiming: Project the 2-D retiming problem onto the **a**-vector and solve this 1-D retiming problem to find the values of $\mathbf{a} \cdot \mathbf{w}_r(e)$ for $e \in E$.

The following subsections describe s-retiming and a-retiming along with the fanout model used in orthogonal 2-D retiming. Throughout these subsections, the notations $x^{(s)}$ and $x^{(a)}$ are used to denote $\mathbf{x} \cdot \mathbf{s}$ and $\mathbf{x} \cdot \mathbf{a}$, respectively.

5.4.1 Fanout Model

In the ILP formulation of 2-D retiming presented in Section 5.3, the fanout constraint is used to ensure that the memory required by the output edges of a node is the maximum memory required by any of the output edges of the node. In 1-D retiming [27], a "gadget" is used to model the fanout node so the memory required by the output edges of the node can be accurately modeled using a linear programming formulation. Figure 5.7 shows a similar gadget used so that the 2-D retiming problem can be modeled as two linear programming problems.

The following four quantities are used in orthogonal 2-D retiming

$$\begin{aligned} w_{max}^{(s)} &= \max_{1 \le i \le k} \left\{ w^{(s)}(e_i) \right\} \\ w_{r,max}^{(s)} &= \max_{1 \le i \le k} \left\{ w_r^{(s)}(e_i) \right\} \\ w_{max}^{(a)} &= \max_{e_i : w_r^{(s)}(e_i) = w_{r,max}^{(s)}} \left\{ w^{(a)}(e_i) \right\} \end{aligned}$$

$$w_{r,max}^{(a)} = \max_{e_i : w_r^{(s)}(e_i) = w_{r,max}^{(s)}} \left\{ w_r^{(a)}(e_i) \right\}.$$

Note that $w_{max}^{(s)}$ are known from the unretimed 2DFG, $w_{r,max}^{(s)}$ and $w_{max}^{(a)}$ are known after s-retiming has been performed, and $w_{r,max}^{(a)}$ are known after s-retiming and a-retiming have been performed.

Figure 5.7(a) shows a fanout node with k output edges. The gadget in Figure 5.7(b) is used to model the fanout node in a 2DFG. Each of the k edges e_i, $1 \leq i \leq k$, has an associated weight $\mathbf{w}(e_i)$ which is known from the 2DFG. The node $\hat{u}$ is a dummy node with zero computation time ($d(\hat{u}) = 0$), and the edges $\hat{e}_i$, $1 \leq i \leq k$, are dummy edges used so the linear programming formulations used in orthogonal 2-D retiming can accurately model the memory required by a node with more than one output edge. We call the edges $\hat{e}_i$, $1 \leq i \leq k$, *auxiliary edges.*

In addition to the weights $\mathbf{w}(e_i)$, each of the edges e_i has the associated quantities $\sigma(e_i) = 1/k$ and

$$\gamma(e_i) = \begin{cases} 1/m & \text{if } w_r^{(s)}(e_i) = w_{r,max}^{(s)} \\ 0 & \text{otherwise} \end{cases},$$

where m is the number of edges e_i satisfying $w_r^{(s)}(e_i) = w_{r,max}^{(s)}$ after s-retiming has been performed. Each auxiliary edge in Figure 5.7(b) has the associated quantities

$$w^{(s)}(\hat{e}_i) = w_{max}^{(s)} - w^{(s)}(e_i)$$

$$w^{(a)}(\hat{e}_i) = w_{max}^{(a)} - w^{(a)}(e_i)$$

and $\sigma(\hat{e}_i) = 1/k$ and

$$\gamma(\hat{e}_i) = \begin{cases} 1/m & \text{if } w_r^{(s)}(\hat{e}_i) = 0 \\ 0 & \text{otherwise} \end{cases},$$

where m has the same definition as it has in $\gamma(e_i)$.

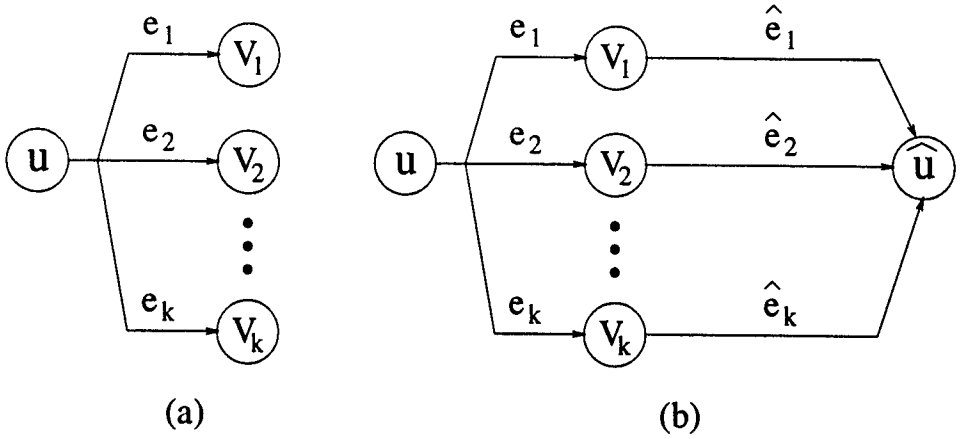


Figure 5.7: (a) A fanout node u. (b) A gadget used to model node u in the linear programming formulations of orthogonal 2-D retiming.

5.4.2 s-Retiming

In orthogonal 2-D retiming, s-retiming affects the memory requirements of the retimed 2DFG more than a-retiming because s-retiming deals with entire delay lines while a-retiming deals with single delays. As a result, s-retiming is performed first on the 2DFG, and then a-retiming is performed.

In s-retiming the 2-D retiming problem is projected onto the scanning vector. Starting with the 2-D retiming equation in (5.1), we can take the dot product of both sides of the equation with the scanning vector $\mathbf{s}$ to get

$$\mathbf{s} \cdot \mathbf{w}_r(e) = \mathbf{s} \cdot \mathbf{w}(e) + \mathbf{s} \cdot \mathbf{r}(v) - \mathbf{s} \cdot \mathbf{r}(u). \tag{5.4}$$

Using the notation $x^{(s)}$ to denote $\mathbf{s} \cdot \mathbf{x}$, (5.4) can be written as

$$w_r^{(s)}(e) = w^{(s)}(e) + r^{(s)}(v) - r^{(s)}(u). \tag{5.5}$$

The first causality constraint in Section 5.3.1 requires $\mathbf{s} \cdot \mathbf{w}_r(e) \geq 0$ for all $e \in E$, which can be rewritten as $w_r^{(s)}(e) \geq 0$ for all $e \in E$. Using this and (5.5) results in

$$w^{(s)}(e) + r^{(s)}(v) - r^{(s)}(u) \geq 0 \tag{5.6}$$

for all $e \in E$. The second causality constraint in Section 5.3.1 and the clock period constraint in Section 5.3.2 are enforced during a-retiming.

The cost function for s-retiming is the total number of access lines crossed by the dependencies. This can be written as

$$COST = \sum_{e \in E} \sigma(e) w_r^{(s)}(e) = \sum_{e \in E} \sigma(e) w^{(s)}(e) + \sum_{e \in E} \sigma(e)(r^{(s)}(v) - r^{(s)}(u)), \qquad (5.7)$$

where $\sigma(e)$ is the weight of an edge according to the fanout model in Section 5.4.1. The formulation of s-retiming consists of minimizing the total number of access lines crossed (i.e., minimize $COST$ in (5.7)) while keeping $w_r^{(s)}(e) \geq 0$ for all $e \in E$ using (5.6).

Since $\sum_{e \in E} \sigma(e) w^{(s)}(e)$ is fixed, s-retiming can be stated as: Minimize

$$COST' = \sum_{v \in V} r^{(s)}(v) \left(\sum_{? \xrightarrow{e} v} \sigma(e) - \sum_{v \xrightarrow{e} ?} \sigma(e) \right)$$

subject to $r^{(s)}(u) - r^{(s)}(v) \leq w^{(s)}(e)$ for all $e \in E$.

Example 5.3 *In this example, we perform s-retiming on the 2DFG in Figure 5.6(a) assuming* $\mathbf{s} = [\ 1 \quad 2\ ]^T$ *and* $\mathbf{a} = [\ -2 \quad 1\ ]^T$. *Using the fanout model described in Section 5.4.1, the 2DFG in Figure 5.6(a) is redrawn in Figure 5.8(a), where node* 5 *is the dummy node associated with fanout node* 1. *The cost function is*

$$\begin{aligned} COST' &= r^{(s)}(1)(1-1) + r^{(s)}(2)\left(\frac{1}{2} - \frac{3}{2}\right) + r^{(s)}(3)\left(\frac{1}{2} - \frac{3}{2}\right) \\ &\quad + r^{(s)}(4)(2-1) + r^{(s)}(5)(1-0) \\ &= -r^{(s)}(2) - r^{(s)}(3) + r^{(s)}(4) + r^{(s)}(5). \end{aligned}$$

The s-retiming problem is to minimize $COST'$ *subject to the following seven causality*

constraints

$$\begin{aligned} r^{(s)}(1) - r^{(s)}(2) &\leq 0 \\ r^{(s)}(1) - r^{(s)}(3) &\leq 0 \\ r^{(s)}(2) - r^{(s)}(4) &\leq 1 \\ r^{(s)}(2) - r^{(s)}(5) &\leq 0 \\ r^{(s)}(3) - r^{(s)}(4) &\leq 3 \\ r^{(s)}(3) - r^{(s)}(5) &\leq 0 \\ r^{(s)}(4) - r^{(s)}(1) &\leq 0, \end{aligned}$$

and the solution found using the linear programming solver in GAMS [63] is $r^{(s)}(1) = 1$, $r^{(s)}(2) = 1$, $r^{(s)}(3) = 3$, $r^{(s)}(4) = 0$, *and* $r^{(s)}(5) = 3$. *The result of s-retiming is shown in Figure 5.8(b), where the numbers in parentheses represent* $w_r^{(s)}(e)$. *This solution is combined with the results of a-retiming in Section 5.4.3 to obtain the complete orthogonal 2-D retiming solution.*

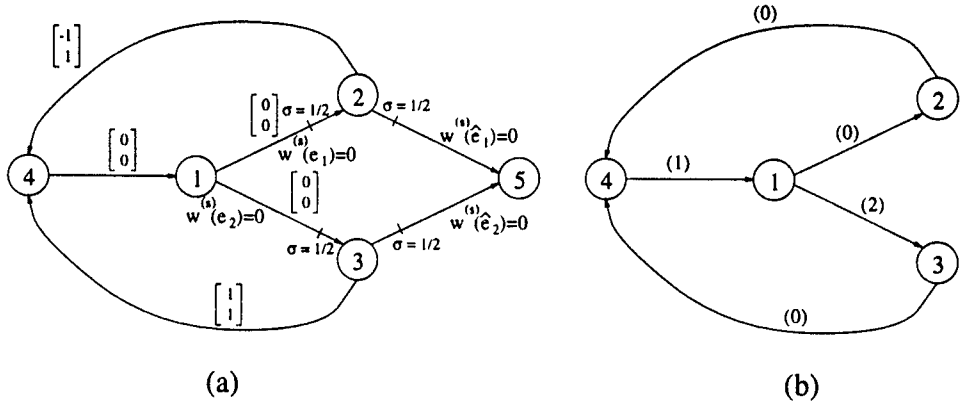


Figure 5.8: (a) The unretimed graph using the fanout model. (b) The result of s-retiming, where the numbers in parentheses represent $w_r^{(s)}(e)$.

The s-retiming formulation accurately models the memory requirements of a fanout node. The following explanation uses the notation introduced in Section 5.4.1. Let the path $u \xrightarrow{e_i} v_i \xrightarrow{\hat{e}_i} \hat{u}$ in Figure 5.7 be denoted as p_i. The values of $w_r^{(s)}(\hat{e}_i)$ are made as small as possible under the constraint $w_r^{(s)}(\hat{e}_i) \geq 0$. Therefore, the value of $r^{(s)}(\hat{u})$ will force $w_r^{(s)}(\hat{e}_i) = 0$ for at least one edge which we call $\hat{e}_j$ (i.e., $w_r^{(s)}(\hat{e}_j) = 0$). Since $\min_{1 \leq i \leq k} \left\{ w_r^{(s)}(\hat{e}_i) \right\} = w_r^{(s)}(\hat{e}_j)$ and the retimed path weights $w_r^{(s)}(p_i)$ are identical for $1 \leq i \leq k$ (they are all equal to $w_{max}^{(s)} + r^{(s)}(\hat{u}) - r^{(s)}(u)$) because the unretimed path

weights $w^{(s)}(p_i)$ are identical (they are all equal to $w^{(s)}_{max}$), we know $w^{(s)}_r(e_j) = w^{(s)}_{r,max}$. This means that

$$w^{(s)}_r(p_j) = w^{(s)}_r(e_j) + w^{(s)}_r(\hat{e}_j) = w^{(s)}_{r,max}.$$

The total cost of the k fanout edges is

$$\begin{aligned}
\sum_{e \in \{e_i, \hat{e}_i\}, 1 \le i \le k} \sigma(e) w^{(s)}_r(e) &= \sum_{e \in \{e_i, \hat{e}_i\}, 1 \le i \le k} \sigma(e) w^{(s)}(e) \\
&\quad + \sum_{e \in \{e_i, \hat{e}_i\}, 1 \le i \le k} \sigma(e) \left(r^{(s)}(v) - r^{(s)}(u) \right) \\
&= k \left(\frac{w^{(s)}_{max}}{k} \right) + \frac{1}{k} \sum_{1 \le i \le k} r^{(s)}(\hat{u}) - \frac{1}{k} \sum_{1 \le i \le k} r^{(s)}(u) \\
&= w^{(s)}_{max} + r^{(s)}(\hat{u}) - r^{(s)}(u) \\
&= w^{(s)}_r(p_j) \\
&= w^{(s)}_{r,max},
\end{aligned}$$

as desired.

5.4.3 a-Retiming

In a-retiming the 2-D retiming problem is projected onto the access vector. While s-retiming takes the first causality constraint of Section 5.3.1 into account, a-retiming takes the second causality constraint and the clock period constraint into account. Like s-retiming, a-retiming is a linear programming formulation which can be solved in polynomial time.

The constraints for a-retiming are the second causality constraint in Section 5.3.1 and the clock period constraint. Starting with (5.1), we can take the dot product of both sides of the equation with the access vector $\mathbf{a}$ to get

$$\mathbf{a} \cdot \mathbf{w}_r(e) = \mathbf{a} \cdot \mathbf{w}(e) + \mathbf{a} \cdot \mathbf{r}(v) - \mathbf{a} \cdot \mathbf{r}(u). \tag{5.8}$$

Using the notation $x^{(a)}$ to denote $\mathbf{a} \cdot \mathbf{x}$, (5.8) can be written as

$$w_r^{(a)}(e) = w^{(a)}(e) + r^{(a)}(v) - r^{(a)}(u). \tag{5.9}$$

The second causality constraint in Section 5.3.1 requires $w_r^{(a)}(e) \geq 0$ for all $e \in E$ such that $w_r^{(s)}(e) = 0$. Using this in (5.9) results in

$$w^{(a)}(e) + r^{(a)}(v) - r^{(a)}(u) \geq 0 \tag{5.10}$$

for all $e \in E$ such that $w_r^{(s)}(e) = 0$.

Clock period constraints must also be taken into account during a-retiming. A set of constraints for a-retiming is formulated such that the clock period of the retimed graph satisfies $\Phi(G_r) \leq c$ for some desired clock period c. The following notations are used:

$$\begin{aligned}
W^{(s)}(u,v) &= \min\{w^{(s)}(p) : u \xrightarrow{p} v\}, \quad u,v \in V \\
W_r^{(s)}(u,v) &= \min\{w_r^{(s)}(p) : u \xrightarrow{p} v\}, \quad u,v \in V \\
W^{(a)}(u,v) &= \min\{w^{(a)}(p) : u \xrightarrow{p} v \text{ and } w_r^{(s)}(p) = W_r^{(s)}(u,v)\}, \quad u,v \in V \\
W_r^{(a)}(u,v) &= \min\{w_r^{(a)}(p) : u \xrightarrow{p} v \text{ and } w_r^{(s)}(p) = W_r^{(s)}(u,v)\}, \quad u,v \in V \\
D(u,v) &= \max\{d(p) : u \xrightarrow{p} v \text{ and } w^{(a)}(p) = W^{(a)}(u,v)\}, \quad u,v \in V \\
D_r(u,v) &= \max\{d_r(p) : u \xrightarrow{p} v \text{ and } w_r^{(a)}(p) = W_r^{(a)}(u,v)\}, \quad u,v \in V
\end{aligned}$$

The following two lemmas are useful for finding a-retiming conditions which satisfy a given clock period constraint.

Lemma 5.4 *Let r be a legal 2-D retiming which retimes G to G_r. The following hold:*

5.4.1 $W_r^{(a)}(u,v) = W^{(a)}(u,v) + r^{(a)}(v) - r^{(a)}(u)$.

5.4.2 $D_r(u,v) = D(u,v)$.

Proof:

(5.4.1)

$$\begin{aligned} W_r^{(a)}(u,v) &= \min\{w_r^{(a)}(p) : u \xrightarrow{p} v \quad \text{and} \quad w_r^{(s)}(p) = W_r^{(s)}(u,v)\} \\ &= \min\{w^{(a)}(p) + r^{(a)}(v) - r^{(a)}(u) : u \xrightarrow{p} v \quad \text{and} \quad w_r^{(s)}(p) = W_r^{(s)}(u,v)\} \\ &= r^{(a)}(v) - r^{(a)}(u) + \min\{w^{(a)}(p) : u \xrightarrow{p} v \quad \text{and} \quad w_r^{(s)}(p) = W_r^{(s)}(u,v)\} \\ &= r^{(a)}(v) - r^{(a)}(u) + W^{(a)}(u,v) \end{aligned}$$

(5.4.2) We can use $d(p) = d_r(p)$ and the result from 5.4.1 to write

$$\begin{aligned} D_r(u,v) &= \max\{d_r(p) : u \xrightarrow{p} v \\ &\qquad \text{and} \quad w^{(a)}(p) + r^{(a)}(v) - r^{(a)}(u) = W^{(a)}(u,v) + r^{(a)}(v) - r^{(a)}(u)\} \\ &= \max\{d(p) : u \xrightarrow{p} v \quad \text{and} \quad w^{(a)}(p) = W^{(a)}(u,v)\} \\ &= D(u,v). \square \end{aligned}$$

Lemma 5.5 *For a legal retiming G_r of G, the following are equivalent:*

5.5.1 $\Phi(G_r) \leq c$.

5.5.2 If $D_r(u,v) > c$ *and* $W_r^{(s)}(u,v) = 0$, *then* $W_r^{(a)}(u,v) \geq \mathbf{a} \cdot \mathbf{a}$.

The proof of Lemma 5.5 is similar to the proof of Lemma 5.2. Lemmas 5.4 and 5.5 are used to prove the following.

Theorem 5.6 *Given an s-retiming solution such that* $r^{(s)}(u) - r^{(s)}(v) \leq w^{(s)}(e)$ *for all edges* $u \xrightarrow{e} v$ *in* E, *the values* $r^{(a)}(v)$ *result in a legal 2-D retiming of* G *such that* $\Phi(G_r) \leq c$ *if and only if*

5.6.1 $r^{(a)}(u) - r^{(a)}(v) \leq w^{(a)}(e)$ *for all* $e \in E$ *such that* $w_r^{(s)}(e) = 0$.

5.6.2 $r^{(a)}(u) - r^{(a)}(v) \leq W^{(a)}(u,v) - \mathbf{a}\cdot\mathbf{a}$ *for all vertices* $u, v \in V$ *such that* $D(u,v) > c$ *and* $W_r^{(s)}(u,v) = 0$.

Proof: 5.6.1 is simply the second causality constraint for a legal 2-D retiming. If 5.6.1 holds, then $\mathbf{r}$ is a legal retiming and by Lemma 5.5 the retimed graph G_r has clock period $\Phi(G_r) \leq c$ under the condition $W_r^{(a)}(u,v) \geq \mathbf{a}\cdot\mathbf{a}$ for all vertices $u, v \in V$ such that $D_r(u,v) > c$ and $W_r^{(s)}(u,v) = 0$. From Lemma 5.4, we know $D_r(u,v) = D(u,v)$ and $W_r^{(a)}(u,v) = W^{(a)}(u,v) + r^{(a)}(v) - r^{(a)}(u)$. Therefore, Lemma 5.5 states that $\Phi(G_r) \leq c$ is equivalent to 5.6.2. □

The cost of a-retiming is the weighted number of scanning lines crossed, given by

$$COST = \sum_{e\in E} \gamma(e) w_r^{(a)}(e) = \sum_{e\in E} \gamma(e) w^{(a)}(e) + \sum_{e\in E} \gamma(e)(r^{(a)}(v) - r^{(a)}(u)).$$

Since $\sum_{e\in E} \gamma(e) w^{(a)}(e)$ is fixed, a-retiming can be stated as follows: Minimize

$$COST' = \sum_{v\in V} r^{(a)}(v) \left(\sum_{?\xrightarrow{e} v} \gamma(e) - \sum_{v\xrightarrow{e} ?} \gamma(e) \right)$$

subject to

1. $r^{(a)}(u) - r^{(a)}(v) \leq w^{(a)}(e)$ for all $e \in E$ such that $w_r^{(s)}(e) = 0$.
2. $r^{(a)}(u) - r^{(a)}(v) \leq W^{(a)}(u,v) - \mathbf{a}\cdot\mathbf{a}$ for all $u, v \in V$ such that $D(u,v) > c$ and $W_r^{(s)}(u,v) = 0$.

Example 5.4 *In this example, a-retiming is performed on the 2DFG in Figure 5.6(a). Since a-retiming depends on the results of s-retiming, the results of s-retiming found in Example 5.3 are used in this example. The 2DFG in Figure 5.6(a) is redrawn in Figure 5.9(a), where the values of* $w^{(a)}(e)$ *and* $w_r^{(s)}(e)$ *are explicitly shown. We assume that the computation time of each node is* 1 *time unit, with the exception that the computation*

Table 5.4: The values of $W_r^{(s)}(u,v)$, $W^{(a)}(u,v)$, and $D(u,v)$ for Example 5.4.

$W_r^{(s)}(u,v)$	1	2	3	4	5
1	0	0	2	0	2
2	1	0	3	0	2
3	1	1	0	0	0
4	1	1	3	0	3
5	-	-	-	-	0

$W^{(a)}(u,v)$	1	2	3	4	5
1	0	0	0	3	0
2	3	0	3	3	0
3	-1	-1	0	-1	0
4	0	0	0	0	0
5	-	-	-	-	0

$D(u,v)$	1	2	3	4	5
1	1	2	2	3	2
2	3	1	4	2	1
3	3	4	1	2	1
4	2	3	3	1	3
5	-	-	-	-	0

time of the dummy node 5 *is zero. The goal is to retime the 2DFG so it can be clocked with a clock period of* 1 *time unit.*

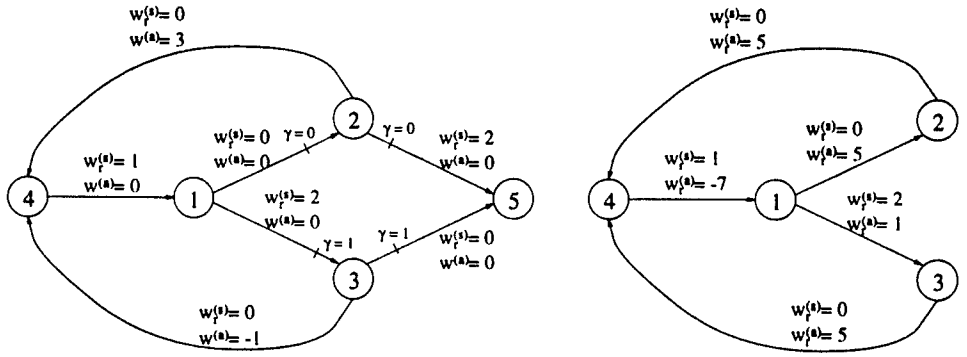


Figure 5.9: (a) The 2DFG which is subjected to a-retiming in Example 5.4. (b) The results of s-retiming and a-retiming for the 2DFG in Figure 5.6(a). These results are found in Examples 5.3 and 5.4.

For fanout node 1, $w_{max}^{(s)} = 0$, $w_{r,max}^{(s)} = 2$, $w_{max}^{(a)} = 0$, *and* $m = 1$. *The values of* $W_r^{(s)}(u,v)$, $W^{(a)}(u,v)$, *and* $D(u,v)$ *are given in Table 5.4.*

The a-retiming formulation is to minimize

$$\begin{aligned} COST' &= r^{(a)}(1)(1-1) + r^{(a)}(2)(0-1) + r^{(a)}(3)(1-2) \\ &\quad + r^{(a)}(4)(2-1) + r^{(a)}(5)(1-0) \\ &= -r^{(a)}(2) - r^{(a)}(3) + r^{(a)}(4) + r^{(a)}(5) \end{aligned}$$

subject to

$$\begin{aligned} r^{(a)}(1) - r^{(a)}(2) &\leq 0 \\ r^{(a)}(2) - r^{(a)}(4) &\leq 3 \\ r^{(a)}(3) - r^{(a)}(4) &\leq -1 \\ r^{(a)}(3) - r^{(a)}(5) &\leq 0 \\ r^{(a)}(1) - r^{(a)}(2) &\leq -5 \\ r^{(a)}(1) - r^{(a)}(4) &\leq -2 \\ r^{(a)}(2) - r^{(a)}(4) &\leq -2 \\ r^{(a)}(3) - r^{(a)}(4) &\leq -6. \end{aligned}$$

The a-retiming solution found using the linear programming solver in GAMS [63] is $r^{(a)}(1) = -7$, $r^{(a)}(2) = -2$, $r^{(a)}(3) = -6$, $r^{(a)}(4) = 0$, *and* $r^{(a)}(5) = -6$. *The 2DFG is drawn in Figure 5.9(b) with the results of s-retiming (from Example 5.3) and a-retiming shown.*

We can show that the a-retiming formulation accurately models the memory requirements of a fanout node when the practical restriction

$$|w_r^{(a)}(e)| < H_{max}(\mathbf{a} \cdot \mathbf{a})/2.$$

is enforced. Assume that $F(\mathbf{w}_r(e))$ is used to estimate the memory required by the edge e.

Lemma 5.7 *If* $w_r^{(s)}(e_i) < w_r^{(s)}(e_j)$, *then* $F(\mathbf{w}_r(e_i)) < F(\mathbf{w}_r(e_j))$.

Proof:

$$w_r^{(s)}(e_i) < w_r^{(s)}(e_j) \quad \Rightarrow \quad w_r^{(s)}(e_i) + 1 \leq w_r^{(s)}(e_j)$$

$$\Rightarrow \quad H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_i) + H_{max}(\mathbf{a}\cdot\mathbf{a}) \leq H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_j)$$

$$\Rightarrow \quad H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_i) + H_{max}(\mathbf{a}\cdot\mathbf{a})/2 \leq H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_j)$$

$$-H_{max}(\mathbf{a}\cdot\mathbf{a})/2 \tag{5.11}$$

Using $w_r^{(a)}(e_i) < H_{max}(\mathbf{a}\cdot\mathbf{a})/2$ and $w_r^{(a)}(e_j) > -H_{max}(\mathbf{a}\cdot\mathbf{a})/2$, we can write the inequalities

$$H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_i) + w_r^{(a)}(e_i) < H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_i) + H_{max}(\mathbf{a}\cdot\mathbf{a})/2$$

and

$$H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_j) + w_r^{(a)}(e_j) > H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_j) - H_{max}(\mathbf{a}\cdot\mathbf{a})/2.$$

Combining these with the inequality in (5.11) results in

$$H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_i) + w_r^{(a)}(e_i) < H_{max}(\mathbf{a}\cdot\mathbf{a})w_r^{(s)}(e_j) + w_r^{(a)}(e_j)$$

$$\Rightarrow \quad F(\mathbf{w}_r(e_i)) < F(\mathbf{w}_r(e_j)).\square$$

The following explanation uses the notation introduced in Section 5.4.1. From Lemma 5.7, we know that for a node u with k output edges, the edge e_j which satisfies $F(\mathbf{w}_r(e_j)) \geq F(\mathbf{w}_r(e_i))$, $1 \leq i \leq k$, must obey $w_r^{(s)}(e_j) = w_{r,max}^{(s)}$ after s-retiming. Given that $w_r^{(s)}(e_j) = w_{r,max}^{(s)}$, from the definition of $F(\cdot)$ we can see that the edge e_j which satisfies $F(\mathbf{w}_r(e_j)) \geq F(\mathbf{w}_r(e_i))$, $1 \leq i \leq k$, also satisfies $w_r^{(a)}(e_j) = w_{r,max}^{(a)}$. To summarize, the edge e_j which satisfies $F(\mathbf{w}_r(e_j)) \geq F(\mathbf{w}_r(e_i))$, $1 \leq i \leq k$, satisfies $w_r^{(s)}(e_j) = w_{r,max}^{(s)}$ and $w_r^{(a)}(e_j) = w_{r,max}^{(a)}$.

The goal now is to show that the cost of the fanout node, given by

$$\sum_{e\in\{e_i,\hat{e}_i\},1\leq i\leq k} \gamma(e)w_r^{(a)}(e),$$

is equal to $w_{r,max}^{(a)}$. Let the path $u \xrightarrow{e_i} v_i \xrightarrow{\hat{e}_i} \hat{u}$ in Figure 5.7(b) be denoted as p_i. The only auxiliary edges which affect the cost function are those with $w_r^{(s)}(\hat{e}_i) = 0$ because $\gamma(\hat{e}_i) =$

0 for any auxiliary edge with $w_r^{(s)}(\hat{e}_i) > 0$. For the auxiliary edges with $w_r^{(s)}(\hat{e}_i) = 0$, the values of $w_r^{(a)}(\hat{e}_i)$ are made as small as possible under the constraint $w_r^{(a)}(\hat{e}_i) \geq 0$. Therefore, the value of $r^{(a)}(\hat{u})$ will force $w_r^{(a)}(\hat{e}_i) = 0$ for at least one edge which satisfies $w_r^{(s)}(\hat{e}_i) = 0$. Let this edge with $w_r^{(a)}(\hat{e}_i) = 0$ and $w_r^{(s)}(\hat{e}_i) = 0$ be the edge $\hat{e}_j$. Since

$$\min_{\hat{e}_i : w_r^{(s)}(\hat{e}_i)=0} \left\{ w_r^{(a)}(\hat{e}_i) \right\} = w_r^{(a)}(\hat{e}_j)$$

and the retimed path weights $w_r^{(a)}(p_i)$ are identical for $1 \leq i \leq k$ (they are all equal to $w_{max}^{(a)} + r^{(a)}(\hat{u}) - r^{(a)}(u)$) because the unretimed path weights $w^{(a)}(p_i)$ are identical (they are all equal to $w_{max}^{(a)}$), we know $w_r^{(a)}(e_j) = w_{r,max}^{(a)}$. This means that

$$w_r^{(a)}(p_j) = w_r^{(a)}(e_j) + w_r^{(a)}(\hat{e}_j) = w_{r,max}^{(a)}.$$

The total cost of the k fanout edges is

$$\begin{aligned}
\sum_{e \in \{e_i, \hat{e}_i\}, 1 \leq i \leq k} \gamma(e) w_r^{(a)}(e) &= \sum_{e \in \{e_i, \hat{e}_i\}, 1 \leq i \leq k} \gamma(e) w^{(a)}(e) \\
&\quad + \sum_{e \in \{e_i, \hat{e}_i\}, 1 \leq i \leq k} \gamma(e) \left(r^{(a)}(v) - r^{(a)}(u) \right) \\
&= m \left(\frac{w_{max}^{(a)}}{m} \right) + \frac{1}{m} \left(m r^{(a)}(\hat{u}) - m r^{(a)}(u) \right) \\
&= w_{max}^{(a)} + r^{(a)}(\hat{u}) - r^{(a)}(u) \\
&= w_r^{(a)}(p_j) \\
&= w_{r,max}^{(a)},
\end{aligned}$$

as desired.

5.4.4 Combining the results of s-retiming and a-retiming

The results of s-retiming and a-retiming must be combined to get the retimed 2DFG. From $w_r^{(s)}(e) = \mathbf{w}_r(e) \cdot \mathbf{s}$ and $w_r^{(a)}(e) = \mathbf{w}_r(e) \cdot \mathbf{a}$, we can write

$$\begin{bmatrix} w_r^{(s)}(e) \\ w_r^{(a)}(e) \end{bmatrix} = \begin{bmatrix} \mathbf{s}^T \\ \mathbf{a}^T \end{bmatrix} \mathbf{w}_r(e),$$

so $\mathbf{w}_r(e)$ can be computed using

$$\mathbf{w}_r(e) = \begin{bmatrix} \mathbf{s}^T \\ \mathbf{a}^T \end{bmatrix}^{-1} \begin{bmatrix} w_r^{(s)}(e) \\ w_r^{(a)}(e) \end{bmatrix}. \tag{5.12}$$

Example 5.5 *For the retiming performed in Examples 5.3 and 5.4, the processing order was specified by* $\mathbf{s} = [\ 1 \quad 2\ ]^T$ *and* $\mathbf{a} = [\ -2 \quad 1\ ]^T$. *Using these values in (5.12) gives*

$$\mathbf{w}_r(e) = \frac{1}{5} \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} w_r^{(s)}(e) \\ w_r^{(a)}(e) \end{bmatrix}.$$

Applying this to the results shown in Figure 5.9(b) gives the retimed 2DFG shown in Figure 5.10, which is the result of applying orthogonal 2-D retiming to the 2DFG in Figure 5.6(a).

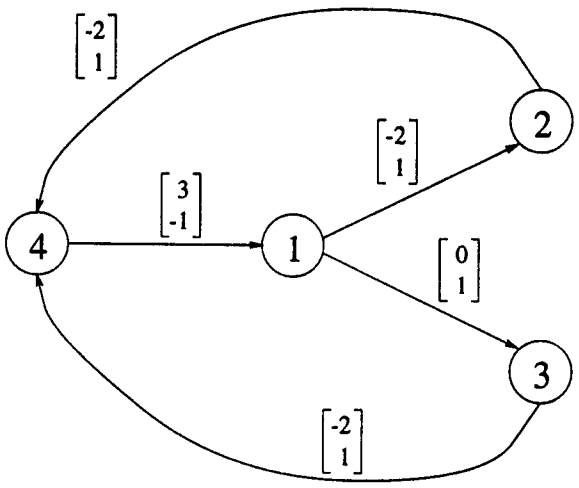


Figure 5.10: The result of performing orthogonal 2-D retiming on the 2DFG in Figure 5.6(a).

A problem with orthogonal 2-D retiming is that **s**-retiming and **a**-retiming may give incompatible results. To show this, we consider an alternative solution to a-retiming in Example 5.4. The solution $r^{(a)}(1) = -8$, $r^{(a)}(2) = -2$, $r^{(a)}(3) = -6$, $r^{(a)}(4) = 0$, and $r^{(a)}(5) = -6$ has the same cost and satisfies all of the a-retiming constraints; however, this new a-retiming solution is not compatible with the s-retiming solution found in

Example 5.3. To see this, note that for the edge $4 \xrightarrow{e} 1$, we found $w_r^{(s)}(e) = 1$ in Example 5.3 and our new solution to a-retiming gives $w_r^{(a)}(e) = 0 + (-8) - 0 = -8$, so the dependency for this edge in the retimed 2DFG is

$$\mathbf{w}(e) = \frac{1}{5}\begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ -8 \end{bmatrix} = \begin{bmatrix} 17/5 \\ -6/5 \end{bmatrix}.$$

Since this dependence vector has non-integer elements, the retimed 2DFG is not practical. The following section introduces a variation of orthogonal 2-D retiming which guarantees that the retimed dependencies have integer elements for a common set of processing orders.

5.5 Integer Orthogonal 2-D Retiming

Integer orthogonal 2-D retiming can be used to guarantee that the edge dependence vectors have integer elements when the scanning vector has the form $\mathbf{s} = \begin{bmatrix} 1 & \mathrm{k} \end{bmatrix}^T$ or $\mathbf{s} = \begin{bmatrix} \mathrm{k} & 1 \end{bmatrix}^T$, where k is a nonnegative integer. Similar to orthogonal 2-D retiming, s-retiming and a-retiming are used in integer orthogonal retiming, but a-retiming is manipulated in integer orthogonal retiming so the dependencies are guaranteed to have integer elements. Since integer orthogonal retiming consists of solving two linear programming problems, it can be solved in polynomial time.

5.5.1 a-retiming for the $s_x = 1$ Case

The first constraint for a-retiming is $r^{(a)}(u) - r^{(a)}(v) \leq w^{(a)}(e)$ for all edges $u \xrightarrow{e} v$ in E such that $w_r^{(s)}(e) = 0$. This can be written as

$$\left(\begin{bmatrix} r_x(u) \\ r_y(u) \end{bmatrix} - \begin{bmatrix} r_x(v) \\ r_y(v) \end{bmatrix}\right) \cdot \mathbf{a} \leq w^{(a)}(e) \tag{5.13}$$

for all $e \in E$ such that $w_r^{(s)}(e) = 0$. From $r^{(s)}(u) = \mathbf{r}(u) \cdot \mathbf{s}$, we know $r_x(u)s_x + r_y(u)s_y = r^{(s)}(u)$, which implies $r_x(u) = r^{(s)}(u) - r_y(u)s_y$ because $s_x = 1$ is assumed. Substituting

this expression for $r_x(u)$ into (5.13) gives

$$\left(\begin{bmatrix} r^{(s)}(u) - r_y(u)s_y \\ r_y(u) \end{bmatrix} - \begin{bmatrix} r^{(s)}(v) - r_y(v)s_y \\ r_y(v) \end{bmatrix} \right) \cdot \mathbf{a} \leq w^{(a)}(e). \tag{5.14}$$

Assuming that $\mathbf{a}$ and $\mathbf{s}$ are related by $a_x = -s_y$ and $a_y = s_x = 1$, (5.14) can be written as

$$-s_y(r^{(s)}(u) - r_y(u)s_y - r^{(s)}(v) + r_y(v)s_y) + (r_y(u) - r_y(v)) \leq w^{(a)}(e). \tag{5.15}$$

Since the first constraint for a-retiming applies to the edges with $w_r^{(s)}(e) = 0$, this implies $w^{(s)}(e) = r^{(s)}(u) - r^{(s)}(v)$, so we can replace $r^{(s)}(u) - r^{(s)}(v)$ with $w^{(s)}(e)$ in (5.15) to get

$$-s_y w^{(s)}(e) + (r_y(u) - r_y(v))(1 + s_y^2) \leq w^{(a)}(e).$$

Expanding $w^{(s)}(e) = s_x w_x(e) + s_y w_y(e)$ and $w^{(a)}(e) = -s_y w_x(e) + s_x w_y(e)$ results in

$$-s_y(s_x w_x(e) + s_y w_y(e)) + (r_y(u) - r_y(v))(1 + s_y^2) \leq -s_y w_x(e) + s_x w_y(e),$$

which can be rewritten using $s_x = 1$ as

$$\begin{aligned} & -s_y w_x(e) - s_y^2 w_y(e) + (r_y(u) - r_y(v))(1 + s_y^2) \leq -s_y w_x(e) + w_y(e) \\ \Rightarrow \quad & (r_y(u) - r_y(v))(1 + s_y^2) \leq w_y(e)(1 + s_y^2) \\ \Rightarrow \quad & r_y(u) - r_y(v) \leq w_y(e). \end{aligned}$$

Therefore, the first constraint for a-retiming when $\mathbf{s} = \begin{bmatrix} 1 & \mathrm{k} \end{bmatrix}^T$ is $r_y(u) - r_y(v) \leq w_y(e)$ for all $e \in E$ such that $w_r^{(s)} = 0$.

The second constraint for a-retiming is $r^{(a)}(u) - r^{(a)}(v) \leq W^{(a)}(u, v) - \mathbf{a} \cdot \mathbf{a}$ for all $u, v \in V$ such that $D(u, v) > c$ and $W_r^{(s)}(u, v) = 0$. Using the left-hand-side of (5.15) to substitute for $r^{(a)}(u) - r^{(a)}(v)$, this can be written as

$$-s_y(r^{(s)}(u) - r^{(s)}(v)) + (r_y(u) - r_y(v))(1 + s_y^2) \leq W^{(a)}(u, v) - \mathbf{a} \cdot \mathbf{a}$$

for all $u, v \in V$ such that $D(u,v) > c$ and $W_r^{(s)}(u,v) = 0$. Solving for $r_y(u) - r_y(v)$, the second constraint for a-retiming can be written as

$$r_y(u) - r_y(v) \leq \frac{W^{(a)}(u,v) - \mathbf{a} \cdot \mathbf{a} + s_y(r^{(s)}(u) - r^{(s)}(v))}{1 + s_y^2}$$

for all $u, v \in V$ such that $D(u,v) > c$ and $W_r^{(s)}(u,v) = 0$. The left-hand side of this inequality must be an integer, but the right-hand side is not guaranteed to be an integer (this occurs in Example 5.6), so we can rewrite this inequality as

$$r_y(u) - r_y(v) \leq \left\lfloor \frac{W^{(a)}(u,v) - \mathbf{a} \cdot \mathbf{a} + s_y(r^{(s)}(u) - r^{(s)}(v))}{1 + s_y^2} \right\rfloor$$

for all $u, v \in V$ such that $D(u,v) > c$ and $W_r^{(s)}(u,v) = 0$.

The cost function for a-retiming is

$$COST' = \sum_{v \in V} r^{(a)}(v) \left(\sum_{? \xrightarrow{e} v} \gamma(e) - \sum_{v \xrightarrow{e} ?} \gamma(e) \right).$$

If we let $k_v = (\sum_{? \xrightarrow{e} v} \gamma(e) - \sum_{v \xrightarrow{e} ?} \gamma(e))$, then the cost can be written as

$$\begin{aligned} COST' &= \sum_{v \in V} (-s_y r_x(v) + r_y(v)) k_v \\ &= \sum_{v \in V} (-s_y (r^{(s)}(v) - r_y(v) s_y) + r_y(v)) k_v \\ &= \sum_{v \in V} (-s_y r^{(s)}(v)) k_v + \sum_{v \in V} r_y(v)(1 + s_y^2) k_v. \end{aligned}$$

During a-retiming, $\sum_{v \in V} (-s_y r^{(s)}(v)) k_v$ and $(1 + s_y^2)$ are constant values, so minimizing $COST'$ is equivalent to minimizing

$$COST'' = \sum_{v \in V} r_y(v) \left(\sum_{? \xrightarrow{e} v} \gamma(e) - \sum_{v \xrightarrow{e} ?} \gamma(e) \right).$$

Summarizing, the a-retiming formulation for the case when $\mathbf{s} = \begin{bmatrix} 1 & \mathrm{k} \end{bmatrix}^T$ is given by: Minimize

$$COST'' = \sum_{v \in V} r_y(v) \left(\sum_{? \xrightarrow{e} v} \gamma(e) - \sum_{v \xrightarrow{e} ?} \gamma(e) \right).$$

subject to

1. $r_y(u) - r_y(v) \leq w_y(e)$ for all $e \in E$ such that $w_r^{(s)}(e) = 0$.

2. $r_y(u) - r_y(v) \leq \left\lfloor \frac{W^{(a)}(u,v) - \mathbf{a}\cdot\mathbf{a} + s_y(r^{(s)}(u) - r^{(s)}(v))}{1+s_y^2} \right\rfloor$ for all $u, v \in V$ such that $D(u,v) > c$ and $W_r^{(s)}(u,v) = 0$.

After solving for the values of $r_y(v)$, the values of $r_x(v)$ can be computed using $r_x(v) = r^{(s)}(v) - r_y(v)s_y$.

Example 5.6 *In this example, we use the integer orthogonal retiming formulation for the case where* $\mathbf{s} = \begin{bmatrix} 1 & k \end{bmatrix}^T$ *to retime the 2DFG shown in Figure 5.11(a) assuming* $\mathbf{s} = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$ *and* $\mathbf{a} = \begin{bmatrix} -1 & 1 \end{bmatrix}^T$. *The desired clock period is 2 units of time, and addition and multiplication are assumed to take 1 and 2 units of time, respectively. The result of s-retiming is shown in Figure 5.11(b), where the numbers on the edges are the values of* $w_r^{(s)}(e)$.

Figure 5.11(c) shows the 2DFG in Figure 5.11(a) with the auxiliary edges included to properly model the fanout of node 1. Since the integer orthogonal retiming formulation uses the values of $w_y(e)$ *for all* $e \in E$, *the values of* $w_y(e)$ *on the auxiliary edges in Figure 5.11(c) are computed using*

$$\mathbf{w}(e) = \begin{bmatrix} \mathbf{s}^T \\ \mathbf{a}^T \end{bmatrix}^{-1} \begin{bmatrix} w^{(s)}(e) \\ w^{(a)}(e) \end{bmatrix}.$$

Then a-retiming consists of minimizing

$$COST'' = r^{(a)}(2) + r^{(a)}(3) - r^{(a)}(4) - r^{(a)}(5) - r^{(a)}(6) + r^{(a)}(7)$$

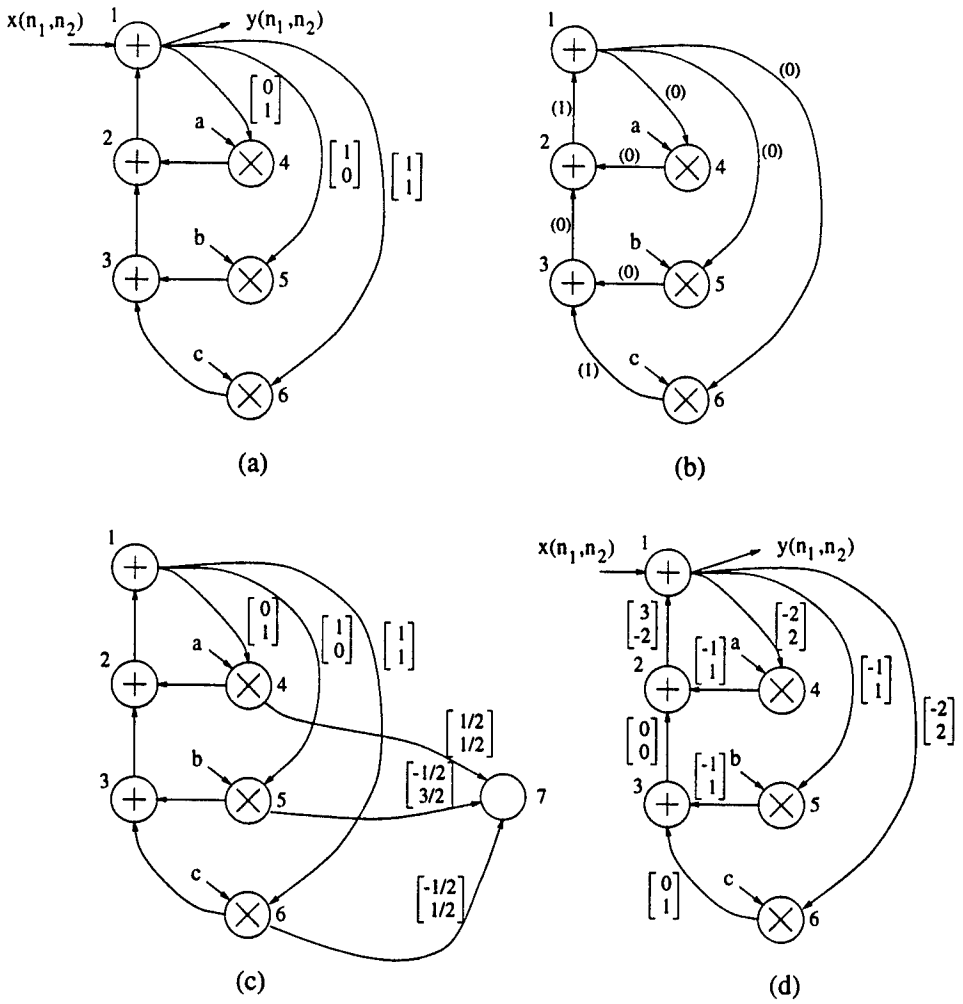


Figure 5.11: (a) The 2DFG which is retimed in Example 5.6. (b) The result of s-retiming. (c) The 2DFG showing the dependencies on the auxiliary edges. (d) The retimed 2DFG which achieves the desired clock period of 2 time units.

subject to the causality constraints

$$\begin{aligned}
r_y(1) - r_y(4) &\leq 1 \\
r_y(1) - r_y(5) &\leq 0 \\
r_y(1) - r_y(6) &\leq 1 \\
r_y(3) - r_y(2) &\leq 0 \\
r_y(4) - r_y(2) &\leq 0 \\
r_y(4) - r_y(7) &\leq 0 \\
r_y(5) - r_y(3) &\leq 0 \\
r_y(5) - r_y(7) &\leq 1 \\
r_y(6) - r_y(7) &\leq 0
\end{aligned}$$

Table 5.5: The values of $W_r^{(s)}(u,v)$, $W^{(a)}(u,v)$, and $D(u,v)$ for Example 5.6.

$W_r^{(s)}(u,v)$	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
2	1	0	1	1	1	1	1
3	1	0	0	1	1	1	1
4	1	0	1	0	1	1	0
5	1	0	0	1	0	1	0
6	2	1	1	2	2	0	0
7	-	-	-	-	-	-	0

$W^{(a)}(u,v)$	1	2	3	4	5	6	7
1	0	1	-1	1	-1	0	1
2	0	0	-1	1	-1	0	1
3	0	0	0	1	-1	0	1
4	0	0	-1	0	-1	0	0
5	0	0	0	1	0	0	2
6	0	0	0	1	-1	0	1
7	-	-	-	-	-	-	0

$D(u,v)$	1	2	3	4	5	6	7
1	1	4	4	3	3	3	3
2	2	1	5	4	4	4	4
3	3	2	1	5	5	5	5
4	4	3	7	2	6	6	2
5	5	4	3	7	2	7	2
6	5	4	3	7	7	2	2
7	-	-	-	-	-	-	0

and the clock period constraints (which use the information in Table 5.5)

$$\begin{aligned}
r_y(1) - r_y(2) &\leq 0 \\
r_y(1) - r_y(3) &\leq -1 \\
r_y(1) - r_y(4) &\leq 0 \\
r_y(1) - r_y(5) &\leq -1 \\
r_y(1) - r_y(6) &\leq 0 \\
r_y(1) - r_y(7) &\leq -1 \\
r_y(4) - r_y(2) &\leq -1 \\
r_y(5) - r_y(2) &\leq -1 \\
r_y(5) - r_y(3) &\leq -1.
\end{aligned}$$

The retimed 2DFG is shown in Figure 5.11(d).

5.5.2 a-retiming for the $s_y = 1$ Case

Using the same techniques as those used in Section 5.5.1 to manipulate a-retiming, we can find that a-retiming has the following formulation when $\mathbf{s} = \begin{bmatrix} \mathrm{k} & 1 \end{bmatrix}^T$.

Minimize

$$COST'' = \sum_{v \in V} (-r_x(v)) \left(\sum_{? \xrightarrow{e} v} \gamma(e) - \sum_{v \xrightarrow{e} ?} \gamma(e) \right).$$

subject to

1. $r_x(u) - r_x(v) \geq w_x(e)$ for all $e \in E$ such that $w_r^{(s)}(e) = 0$.

2. $r_x(u) - r_x(v) \geq \left\lceil \frac{-W^{(a)}(u,v) + \mathbf{a} \cdot \mathbf{a} + s_x(r^{(s)}(u) - r^{(s)}(v))}{1 + s_x^2} \right\rceil$ for all $u, v \in V$ such that $D(u,v) > c$ and $W_r^{(s)}(u,v) = 0$.

After solving for the values of $r_x(v)$, the values of $r_y(v)$ can be computed using $r_y(v) = r^{(s)}(v) - r_x(v)s_x$.

5.6 Comparisons

In this section we compare the results of using our ILP 2-D retiming technique and our orthogonal 2-D retiming technique with the previously published chained [34] and schedule-based [33] 2-D retiming approaches.

Comparisons for the 2DFGs in Figure 5.6(a) and Figure 5.13(a) are given in Table 5.6 and Table 5.7, respectively. The results in these tables assume that the computation time of each node is one time unit, the desired clock period is one time unit, and the 2DFG operates on a 256×256 data set. Because the number of registers required by the retimed 2DFG is not the same for each of the 256^2 iterations, the number of registers required by the retimed 2DFGs is determined by computing the memory required for each of the 256^2 iterations and taking the maximum of these values. To demonstrate this, the memory requirement for the 2DFG in Figure 5.12(a) is computed assuming a 4×4 data set and processing order specified by $\mathbf{s} = [\ 1 \quad 1\ ]^T$ and $\mathbf{a} = [\ \text{-}1 \quad 1\ ]^T$. At the beginning of iteration $[\ 1 \quad 2\ ]^T$, the four samples which must be stored due to the dependency

$[\ 1 \quad 0\ ]^T$ are indicated in Figure 5.12(b) with an "x" and the one sample which must be stored due to the dependency $[\ -1 \quad 1\ ]^T$ is indicated with an "o". Therefore, the iteration $[\ 1 \quad 2\ ]^T$ requires that 5 samples are stored. The reader can verify that the iteration $[\ 1 \quad 1\ ]^T$ requires that 4 samples are stored, the iteration $[\ 2 \quad 2\ ]^T$ requires that 5 samples are stored. The maximum number of samples that must be stored for any iteration is 5, so this 2DFG requires 5 registers.

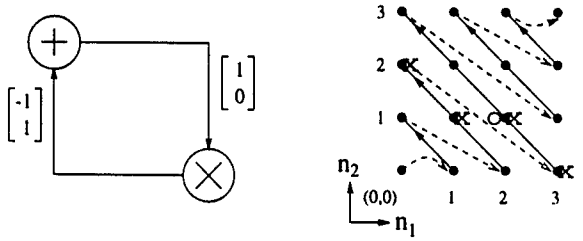


Figure 5.12: (a) A 2DFG. (b) The samples which must be stored.

Because the 2DFG in Figure 5.6(a) is small, the ILP 2-D retiming technique described in Section 5.3 was used to obtain the results in Table 5.6. Note that the minimum length scanning vector feasible for this DFG with schedule-based retiming is $\mathbf{s} = [\ 1 \quad 4\ ]^T$. Due to the relatively large size of the 2DFG in Figure 5.13(a), the orthogonal 2-D retiming technique in Section 5.4 was used to obtain the results in Table 5.7. Since orthogonal 2-D retiming resulted in dependence vectors with integer elements, it was not necessary to use integer orthogonal retiming for this 2DFG. Figure 5.6(b) shows the retimed version of the 2DFG in Figure 5.6(a) for $\mathbf{s} = [\ 1 \quad 2\ ]^T$ and $\mathbf{a} = [\ -2 \quad 1\ ]^T$, and Figure 5.13(b) shows the retimed version of the 2DFG in Figure 5.13(a) for $\mathbf{s} = [\ 1 \quad 1\ ]^T$ and $\mathbf{a} = [\ 1 \quad 1\ ]^T$.

From Tables 5.6 and 5.7, we can observe that the "schedule-based" retiming technique in [33] does not find a solution for any of the processing orders chosen. This is because our techniques have less stringent (but still sufficient) causality constraints than the

Table 5.6: Memory requirements after retiming the circuit in Figure 5.6(a) assuming a 256×256 data set.

scanning vector	retiming technique	number of registers
$\mathbf{s} = [\ 0 \quad 1\ ]^T$	ours	258
	chained	510
	schedule-based	no solution
$\mathbf{s} = [\ 1 \quad 2\ ]^T$	ours	385
	chained	511
	schedule-based	no solution

Table 5.7: Memory requirements after retiming the circuit in Figure 5.13(a) assuming a 256×256 data set.

scanning vector	retiming technique	number of registers
$\mathbf{s} = [\ 1 \quad 2\ ]^T$	ours	778
	chained	1794
	schedule-based	no solution
$\mathbf{s} = [\ 1 \quad 1\ ]^T$	ours	1032
	chained	2048
	schedule-based	no solution
$\mathbf{s} = [\ 2 \quad 1\ ]^T$	ours	780
	chained	1288
	schedule-based	no solution

schedule-based technique. Thus, our techniques are compatible with more processing orders. We can conclude that our techniques offer more flexibility than the schedule-based retiming technique because our techniques are compatible with more processing orders.

We can also conclude from Tables 5.6 and 5.7 that our techniques result in solutions which require considerably less memory than the chained retiming technique in [34]. This is because our formulations are not sensitive to the memory requirements of the unretimed 2DFG, while the results of chained retiming are dependent on the memory requirements of the unretimed 2DFG.

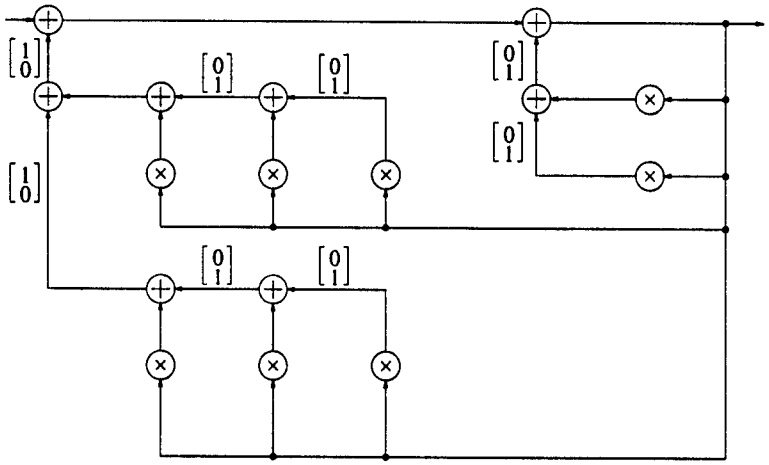
5.7 Conclusions

In this chapter we have presented two techniques for retiming 2DFGs. These two techniques attempt to minimize the amount of memory required to implement the 2DFGs under a given clock period constraint. The ILP 2-D retiming technique solves the entire 2-D retiming problem as a whole but requires long run times to solve. As a result, this technique should be used only for small 2DFGs. Orthogonal 2-D retiming runs faster than the ILP technique but occasionally gives incompatible results between s-retiming and a-retiming. Therefore, orthogonal 2-D retiming should be used when the 2DFG is too large to solve using ILP 2-D retiming, and integer orthogonal 2-D retiming should be used when orthogonal 2-D retiming gives incompatible results between s-retiming and a-retiming.

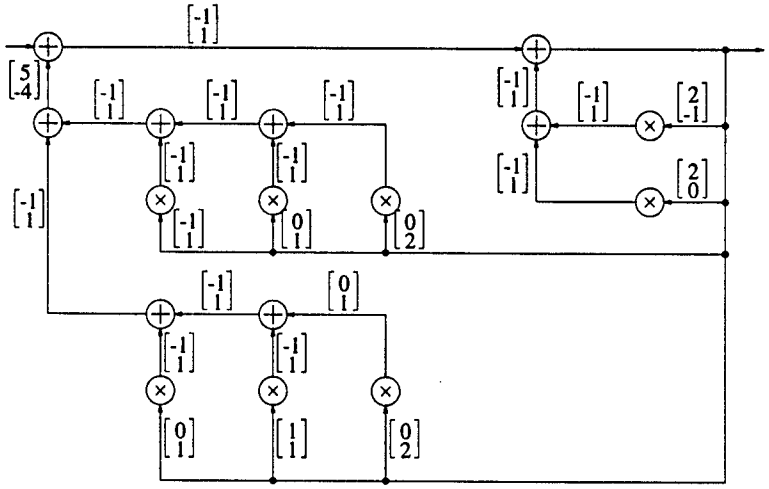
Our comparisons have shown that the techniques presented in this chapter give considerably better results than previously published techniques. In fact, our techniques can result in retimed 2DFGs which require less than 50% of the memory hardware required

by the technique in [34]. Our techniques perform better than the technique in [33] because our formulations have less stringent (but still sufficient) causality constraints, and they perform better than chained retiming in [34] because our formulations are not sensitive to the memory requirements of the unretimed 2DFG, while the results of chained retiming are dependent on the memory requirements of the unretimed 2DFG.

Future research should be directed toward studying the interactions between inter-iteration parallelism and inter-operation parallelism and toward finding algorithms for retiming data-flow graphs which operate on signals which have dimensionality greater than two for applications such as video processing. Register minimization in 2-D retiming which includes the use of scanning order conversion requires further study. Retiming for folding for the one-dimensional case has been studied in [28]. Two-dimensional retiming for folding of 2DFGs is another topic of further research.



(a)



(b)

Figure 5.13: (a) A 2-D IIR filter. (b) A retimed version of the filter.

Chapter 6

Conclusions and Future Research Directions

6.1 Conclusions

We have considered several formal techniques for mapping DSP algorithms to VLSI architectures. The salient features of these techniques are that they increase the understanding of the interaction between algorithms and architectures, and they provide methods for designing new and improved architectures for a wide variety of DSP algorithms.

A new formulation of scheduling was presented in Chapter 2. Using this formulation, we showed that retiming is a special case of scheduling, and we described the interaction between retiming and scheduling in a mathematical framework. Algorithms were developed for exhaustively generating all retiming and scheduling solutions for a strongly connected DFG. By carefully choosing the examples in this chapter, we have given scheduling solutions for many filters which are of interest to the high-level synthesis community. This community should find the scheduling results for the biquad filter and the fifth order wave digital elliptic filter to be of particular interest.

New expressions were introduced in Chapter 3 for computing the minimum number of registers required to implement a statically scheduled DFG. Two cases are considered, namely, the cases where retiming is and is not allowed after the DFG has been scheduled. These results should be useful in CAD tools used to design memory-efficient architectures.

The multirate folding transformation was developed in Chapter 4. Within the scope of multirate folding, the problems of retiming for multirate folding and register minimization in (multirate) folded architectures were also considered. Together, the formulations of multirate folding, retiming for multirate folding, and register minimization provide a new technique for designing single-rate VLSI architectures for multirate DSP algorithms, such as the discrete wavelet transform.

In Chapter 5, two techniques for 2-D retiming were presented, namely, ILP 2-D retiming and orthogonal 2-D retiming. These techniques can reduce the memory usage in 2-D DSP implementations by over 50%. This is of particular importance due to the recent high demand for low cost and low power implementations of 2-D DSP for multimedia applications.

6.2 Future Research Directions

The work presented in this thesis provides the foundation for several interesting future research projects. In the area of exhaustive scheduling and retiming, it would be interesting to include unfolding [62] in the formulation. Since a formulation is given in Chapter 2 for folding, it seems natural that a similar formulation can be derived for unfolding, since unfolding is essentially the inverse operation of folding. A formulation which includes retiming, folding, and unfolding would be interesting from a theoretical

point of view as well as a practical point of view.

In the area of register minimization, we have solved the problem of computing the number of registers required by a scheduled DSP algorithm, but the problem of allocating data to these registers is an open problem. Although several excellent heuristic techniques have been suggested (e.g., in [51], [52], and [53]), the topic of memory management will be an open problem for many years due to the large percentage of chip area which must be dedicated to memory.

In the area of multirate synthesis, the topics of retiming [35] and scheduling [55] for multirate DFGs are still under examination. The study of these topics and the development of formulations for retiming and scheduling similar to those in Chapter 2 (but for the multirate case) would be both useful and interesting.

In the area of multi-dimensional retiming, 2-D retiming with non-linear scanning orders, such as the Dovetail scan [72], would be an interesting extension. Future research should also take into account the cost of scan conversion buffers, i.e., the buffers required to convert the data to and from the traditional line-by-line scanning order. Another area of future research is to extend the 2-D retiming formulations to higher dimensions. This problem, which is by no means trivial, has applications in the very popular area of digital video processing.

Finally, one research topic, which we have not been able to address, includes most of the topics covered in this thesis. This topic is to combine 2-D retiming, multirate folding, and register minimization to develop a multirate/multi-dimensional folding transformation. Such a transformation would be useful for designing new two-dimensional discrete wavelet transform architectures [73] [74].

Bibliography

[1] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley, second ed., 1993.

[2] W. Wolf, *Modern VLSI Design: A Systems Approach.* Englewood Cliffs, NJ: Prentice Hall, second ed., 1994.

[3] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[4] J. S. Lim, *Two-Dimensional Signal and Image Processing.* Englewood Cliffs, NJ: Prentice Hall, 1990.

[5] P. P. Vaidyanathan, *Multirate Systems and Filter Banks.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[6] S. Haykin, *Adaptive Filter Theory.* Englewood Cliffs, NJ: Prentice Hall, second ed., 1991.

[7] M. C. McFarland, A. C. Parker, and R. Composano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, pp. 301–318, February 1990.

[8] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, pp. 356–370, March 1988.

[9] T.-F. Lee *et al.*, "An effective methodology for functional pipelining," in *Proceedings of the Int. Conf. on Computer Aided Design*, pp. 230–233, November 1992.

[10] P. Lippens *et al.*, "PHIDEO: A silicon compiler for high speed algorithms," in *Proceedings of the European Conference on Design Automation*, (Amsterdam), pp. 436–441, February 1991.

[11] M. Potkonjak and J. Rabaey, "Retiming for scheduling," in *VLSI Signal Processing IV*, pp. 23–32, November 1990.

[12] L.-F. Chao, A. LaPaugh, and E. H. Sha, "Rotation scheduling: A loop pipelining algorithm," in *Proceedings of the 30th Design Automation Conference*, pp. 566–572, June 1993.

[13] M. Potkonjak and J. Rabaey, "Pipelining: Just another transformation," in *Proceedings of 1994 IEEE International Conference on Application-Specific Array Processors*, (Oakland, CA), pp. 163–177, August 1992.

[14] C. H. Gebotys and M. I. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1266–1278, September 1993.

[15] S. M. Heemstra de Groot, S. H. Gerez, and O. E. Herrmann, "Range chart guided iterative data-flow graph scheduling," *IEEE Transactions on Circuits and Systems–I: Fundamental Theory and Applications*, vol. 39, pp. 351–364, May 1992.

[16] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 661–679, June 1989.

[17] H. De Man *et al.*, "Cathedral II: A silicon compiler for digital signal processing," *IEEE Design and Test Magazine*, pp. 13–25, December 1986.

[18] H. De Man *et al.*, "Architecture driven synthesis techniques for VLSI implementation of DSP algorithms," *Proceedings of the IEEE*, pp. 319–335, February 1990.

[19] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man, *High-Level Synthesis for Real-Time Digital Signal Processing.* Kluwer Academic, 1993.

[20] I.-C. Park and C.-M. Kyung, "FAMOS: An efficient scheduling algorithm for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1437–1448, October 1993.

[21] R. I. Hartley and J. R. Jasica, "Behavioral to structural translation in a bit-serial silicon compiler," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, pp. 877–886, August 1988.

[22] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, pp. 464–475, April 1991.

[23] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin, "PLS: A scheduler for pipeline synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1279–1286, September 1993.

[24] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of data-path intensive architectures," *IEEE Design and Test of Computers*, pp. 40–51, June 1991.

[25] M. Potkonjak and J. Rabaey, "Fast implementation of recursive programs using transformations," in *Proc. of IEEE Int. Conf on Acoustics, Speech, and Signal Processing*, vol. V, (San Francisco, CA), pp. 569–572, March 1992.

[26] C.-Y. Wang and K. K. Parhi, "High-level DSP synthesis using concurrent transformations, scheduling, and allocation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 274–295, March 1995.

[27] C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming," *Third Caltech Conference on VLSI*, pp. 87–116, 1983.

[28] K. K. Parhi, C.-Y. Wang, and A. P. Brown, "Synthesis of control circuits in folded pipelined DSP architectures," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 29–43, January 1992.

[29] K. K. Parhi, "Calculation of minimum number of registers in arbitrary life time chart," *IEEE Transactions on Circuits and Systems–II: Analog and Digital Signal Processing*, vol. 41, pp. 434–436, June 1994.

[30] S. Simon, E. Bernard, M. Sauer, and J. Nossek, "A new retiming algorithm for circuit design," in *Proceedings of IEEE ISCAS*, (London, England), May 1994.

[31] T. C. Denk and K. K. Parhi, "A unified framework for characterizing retiming and scheduling solutions," in *Proceedings of IEEE ISCAS*, vol. 4, (Atlanta, GA), pp. 568–571, May 1996.

[32] T. C. Denk, M. Majumdar, and K. K. Parhi, "Two-dimensional retiming with low memory requirements," in *Proceedings of IEEE ICASSP*, vol. 6, (Atlanta, GA), pp. 3330–3333, May 1996.

[33] N. L. Passos, E. H.-M. Sha, and S. C. Bass, "Optimizing DSP flow graphs via schedule-based multidimensional retiming," *IEEE Transactions on Signal Processing*, vol. 44, pp. 150–155, January 1996.

[34] N. Passos and E. H.-M. Sha, "Full parallelism in uniform nested loops using multi-dimensional retiming," in *Proc. Int'l Conf. on Parallel Processing*, 1994.

[35] V. Živojnović and R. Schoenen, "On retiming of multirate DSP algorithms," in *Proc. of IEEE Int. Conf on Acoustics, Speech, and Signal Processing*, vol. 6, (Atlanta, GA), pp. 3310–3313, May 1996.

[36] T. C. Denk and K. K. Parhi, "Systematic design of architectures for M-ary tree-structured filter banks," in *VLSI Signal Processing, VIII* (T. Nishitani and K. Parhi, eds.), pp. 157–166, IEEE Press, October 1995.

[37] K. Ito and K. K. Parhi, "Register minimization in cost-optimal synthesis of DSP architectures," in *VLSI Signal Processing, VIII* (T. Nishitani and K. Parhi, eds.), pp. 207–216, IEEE Press, October 1995.

[38] L. E. Lucke and K. K. Parhi, "Data-flow transformations for critical path time reduction in high-level DSP synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1063–1068, July 1993.

[39] C. H. Gebotys and M. I. Elmasry, "Optimal synthesis of high-performance architectures," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 389–397, March 1992.

[40] C. H. Gebotys, "Synthesizing embedded speed optimized architectures," *IEEE Journal of Solid-State Circuits*, vol. 28, pp. 242–252, March 1993.

[41] I. Daubechies, "Orthonormal bases of compactly supported wavelets," *Comm. in Pure and Applied Math.*, vol. 41, pp. 909–996, November 1988.

[42] S. G. Mallat, "Multifrequency channel decompositions of images and wavelet models," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, pp. 2091–2110, December 1989.

[43] O. Rioul and M. Vetterli, "Wavelets and signal processing," *IEEE Signal Processing Magazine*, pp. 14–38, October 1991.

[44] G. Strang, "Wavelets and dilation equations: A brief introduction," *SIAM Rev.*, vol. 31, pp. 614–627, December 1989.

[45] R. Coifman and M. Wickerhauser, "Entropy-based algorithms for best basis selection," *IEEE Transactions on Information Theory*, vol. 38, pp. 713–718, March 1992.

[46] T. C. Denk and K. K. Parhi, "Lower bounds on memory requirements for statically scheduled DSP programs," to appear in *Journal of VLSI Signal Processing*, June 1996.

[47] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and pipelined VLSI implementation of signal processing algorithms," in *VLSI and Modern Signal Processing* (Kung, Whitehouse, and Kailath, eds.), ch. 15, pp. 257–276, Prentice Hall, 1985.

[48] J. Monteiro, S. Devadas, and A. Ghosh, "Retiming sequential circuits for low power," in *Proceedings of IEEE Int. Conf. on Computer Aided Design*, pp. 398–402, 1993.

[49] D. Johnson and J. Johnson, *Graph Theory With Engineering Applications.* New York, NY: The Ronald Press Company, 1972.

[50] R. I. Hartley and K. K. Parhi, *Digit-Serial Computation.* Kluwer Academic, 1995.

[51] K. K. Parhi, "Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 39, pp. 423–440, July 1992.

[52] L. Stok and J. Jess, "Foreground memory management in data path synthesis," *Interational Journal of Circuit Theory and Applications*, vol. 20, pp. 235–255, 1992.

[53] J. Bae, V. Prasanna, and H. Park, "Synthesis of a class of data format converters with specified delays," in *Proceedings of 1994 IEEE International Conference on Application-Specific Array Processors*, (San Francisco, CA), pp. 283–294, IEEE Computer Society Press, August 1994.

[54] F. Kurdahi and A. Parker, "REAL: A program for register allocation," in *Proc. 24th ACM/IEEE Design Automation Conf.*, pp. 210–215, June 1987.

[55] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Computer*, vol. C-36, pp. 24–35, January 1987.

[56] G. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved dataflow programs for DSP computation," in *Proceedings of IEEE ICASSP*, vol. V, (San Francisco, CA), pp. 561–564, March 1992.

[57] S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms," *IEEE Transactions on Signal Processing*, vol. 42, pp. 1190–1201, May 1994.

[58] R. Govindarajan, G. Gao, and P. Desai, "Minimizing memory requirements in rate-optimal schedules," in *Proceedings of IEEE Int. Conf. on Application-Specific Array Processors*, (San Francisco, CA), pp. 75–86, August 1994.

[59] M. Renfors and Y. Neuvo, "Fast multiprocessor realizations of digital filters," in *Proceedings of IEEE ICASSP*, pp. 916–919, 1980.

[60] M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under hardware speed constraints," *IEEE Transactions on Circuits and Systems*, vol. CAS-28, pp. 196–202, March 1981.

[61] J.-G. Chung and K. K. Parhi, "Pipelining of lattice IIR digital filters," *IEEE Transactions on Signal Processing*, vol. 42, pp. 751–761, April 1994.

[62] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Transactions on Computers*, vol. 40, pp. 178–195, February 1991.

[63] A. Brooke, D. Kendrick, and A. Meeraus, *GAMS: A User's Guide, Release 2.25.* South San Francisco, CA: The Scientific Press, 1992.

[64] S. S. Bhattacharyya, J. T. Buck, and E. A. Lee, "A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs," in *Proceedings of IEEE VLSI Signal Processing Workshop*, (Veldhoven, The Netherlands), pp. 188–196, October 1993.

[65] R. Govindarajan and G. R. Gao, "A novel framework for multi-rate scheduling in DSP applications," in *Proceedings of IEEE Int. Conf. on Application-Specific Array Processors*, pp. 77–88, 1993.

[66] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing memory requirements in rate-optimal schedules," in *Proceedings of IEEE Int. Conf. on Application-Specific Array Processors*, (San Francisco, CA), pp. 75–86, August 1994.

[67] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 473–484, April 1992.

[68] G. Goosens, J. Rabaey, J. Vandewalle, and H. De Man, "An efficient microcode compiler for application specific DSP processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 925–937, September 1990.

[69] L. Lamport, "The parallel execution of DO loops," *Communications of the ACM*, vol. 17, pp. 83–93, February 1974.

[70] K. K. Parhi and D. G. Messerschmitt, "Concurrent architectures for two-dimensional recursive digital filtering," *IEEE Transactions on Circuits and Systems*, vol. 36, pp. 813–829, June 1989.

[71] M. Winzker, K. Grüger, W. Gehrke, and P. Pirsch, "VLSI chip set for 2D HDTV subband filtering with on-chip line memories," *IEEE Journal of Solid-State Circuits*, vol. 28, pp. 1354–1361, December 1993.

[72] E. Patrick, D. Anderson, and F. Bechtel, "Mapping multidimensional space to one dimension for computer output display," *IEEE Transactions on Computers*, vol. C-17, pp. 949–953, October 1968.

[73] K. K. Parhi and T. Nishitani, "VLSI architectures for discrete wavelet transforms," *IEEE Transactions on VLSI Systems*, vol. 1, pp. 191–202, June 1993.

[74] C. Chakrabarti and M. Vishwanath, "Efficient realizations of the discrete and continuous wavelet transforms: From single chip implementations to mappings on SIMD array computers," *IEEE Transactions on Signal Processing*, vol. 43, pp. 759–771, March 1995.